Escuela Politécnica Superior

Videojuego RPG en Unity

Grado en Ingeniería Multimedia

Trabajo Fin de Grado

Autor:
Rafael Navarro Estrella

Tutor/es:
Mireia Luisa Sempere Tortosa

Universitat d'Alacant
Universidad de Alicante

Septiembre 2018

Justificación y objetivos

Desde que tenía cuatro años, cuando mis padres nos compraron a mi hermano y a mí una *Game Boy Color*, he vivido jugando a videojuegos de todos los géneros: de lucha, de carreras, de plataformas, etc. Sin embargo, hay uno que siempre me ha tenido enamorado: los juegos de rol o RPG (*Role Playing Game*) que inmergen al jugador en un mundo totalmente diferente y los inscriben dentro de un personaje que los representará dentro de este mundo. Juegos míticos como *The Legend Of Zelda*, la saga *Diablo*, *World of Warcraft*, *The Witcher* o la saga *The Elder Scrolls* pertenecen a este género, entre muchos otros.

Por eso, cuando tuve la oportunidad de elegir el tema de mi Trabajo de Fin de Grado, decidí aprovechar la oportunidad para, por fin, rendir un digno homenaje a este género que, como a mí, ha enamorado y seguirá enamorando a las masas. Y digo por fin porque el deseo de realizar un videojuego de este tipo arde en mi interior desde el segundo cuatrimestre de mi primer año en el grado, hace ahora cinco años.

En aquel entonces, cansado de lo poco que Ingeniería Multimedia parecía estar orientada a videojuegos, decidí hacer yo mismo el mío propio. Mi primera toma de contacto con este mundillo fue un sencillo juego de exploración, programado en *GameMaker 8*. Poco después, y, con la ayuda de tutoriales en *YouTube* aprendí como exprimir aún más las capacidades de este motor y conseguí hacer un ARPG (*Action Role Playing Game*) muy básico y parecido a los juegos *The Legend of Zelda* clásicos. Seguí desarrollando este prototipo, incluyendo nuevas características y mecánicas, a la vez que mejoraba tanto mis habilidades de programación como artísticas.

Desafortunadamente, a principios del año 2017 empecé a encontrarme con una barrera: *GameMaker Studio* está severamente limitado, especialmente en lo relativo a la interfaz. Así pues, en junio, decidí dar el salto de GameMaker a Unity, de GML (*GameMaker Language*) a C#, para aprender a utilizar este motor tan extendido en la actualidad. Así, una vez supe manejar Unity, me propuse el objetivo de reconstruir desde cero ese prototipo y extenderlo, para hacer, por fin, el juego de mis sueños, que rinda un sentido homenaje al género RPG y que pueda presentar orgulloso ante el tribunal.

Agradecimientos

A todos los profesores del grado de ingeniería multimedia que me han enseñado todo lo que sé y me han impulsado a extender mis conocimientos más allá de sus enseñanzas.

A Rafael, Gonzalo, Eduardo y Sergio. Por haber estado junto a mí todos estos años y motivarme para hacer videojuegos.

Dedicatoria

A mis padres, Juan Carlos y Rosa María, por apoyarme en la elección de un camino profesional que para su generación sería impensable.

Citas

El conocimiento es inútil si no se usa.

Mayordomo Gixx, líder del priorato de Durmand
(Guild Wars 2)

Índice de contenido

Contenido

Índice de figuras

Índice de tablas

Introducción

Hace años que el mundo entró en la era digital, y, con ella, la época dorada del entretenimiento digital [1, 2]. En este grupo se inscriben los videojuegos, que podemos clasificar de manera general en dos grandes apartados: para *smartphones* o *tablets*, juegos más sencillos, pero de los que puede haber decenas instalados en un solo dispositivo y a los que el usuario dedicará horas de su día de manera religiosa y juegos para consola u ordenador, más complejos de realizar, pero que transmiten experiencias únicas e importantes lecciones.

A este segundo grupo pertenecen generalmente los juegos RPG, que transmiten al jugador la sensación de formar parte del mundo además de aquellas que el desarrollador tenga la intención de hacer sentir, como puede ser heroicidad o terror. El objetivo de este TFG es crear un juego de este tipo y que aplique estos principios de trasladar fielmente al jugador a un mundo virtual.

Marco teórico o estado del arte

En este apartado se hará un estudio acerca de los videojuegos, particularmente de aquellos del género RPG. Se estudiará y comparará, además, los diversos motores de desarrollo de videojuegos. Por último, se harán observaciones sobre algunos de los problemas que surgen a la hora de implementar un juego de este estilo y como otros juegos del mercado los abordan, para así decidir cuál es el mejor curso de acción.

¿Qué es un videojuego RPG?

Un videojuego RPG, como ya se ha mencionado anteriormente, es un videojuego basado en los juegos de rol de mesa o *tabletop*, en los que el jugador toma el papel o rol de un personaje dentro de un mundo y actúa a través de ese avatar para solucionar o superar los problemas de dicho mundo.

Historia de los RPG

El origen de estos juegos se remonta al año 1982, cuando *Dragonstomper* fue lanzado para la Atari 2600. Un año más tarde, *Bokosuka Wars* haría su debut en la Sharp X1, que sería adaptado dos años más tarde para la consola Sharp X68000 como *New Bokosuka Wars* el cual sentaría las bases de dos subgéneros de los RPG: los SRPG y los ARPG, los cuales se explicarán a

continuación. Finalmente, en 1986, *Chunsoft* creó para NES un juego llamado *Dragon Quest* el cual asienta las bases de los JRPG y es considerado un referente en este género [3].

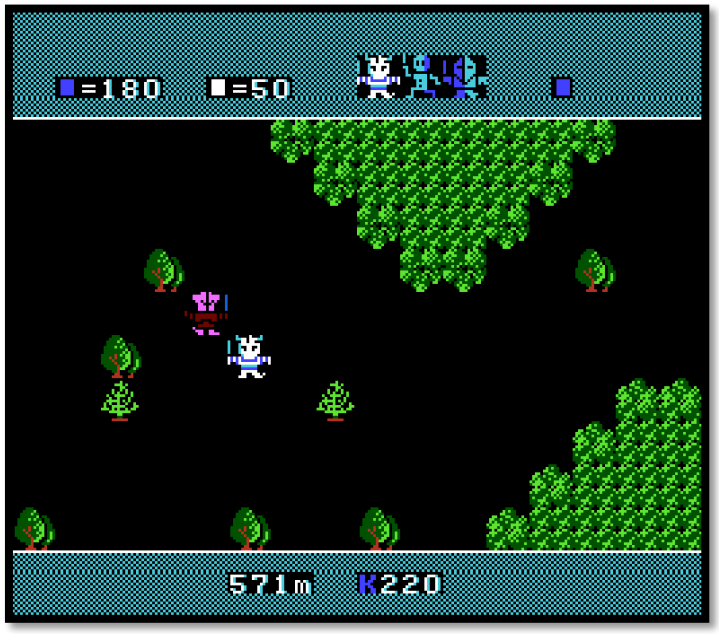


Figura 1 Bokosuka Wars (1983)

Fuente: www.giantbomb.com

Figura 2 Dragon Quest (1986)

Fuente: www.reasearchgate.net

Tipos de RPG

El término RPG es muy amplio y por eso, abarca diferentes tipos de subgéneros, todos similares, pero con claras diferencias entre ellos. Los principales son:

ARPG

Los juegos ARPG (*Action Role Playing Game*) son aquellos que, aunque tienen el tipo de progresión típico de un RPG, combinan este con un combate en tiempo real, con habilidades que favorecen la reacción del jugador como esquivar proyectiles o bloquear ataques con un escudo. Juegos famosos de este género son *Diablo* o *The Legend of Zelda*. Títulos recientes que se inscriben bajo este género son *The Binding of Isaac* o *Hyper Light Drifter*.

Figura 3 The Legend of Zelda: A Link to the Past (1991)

Fuente: www.ultimagame.es

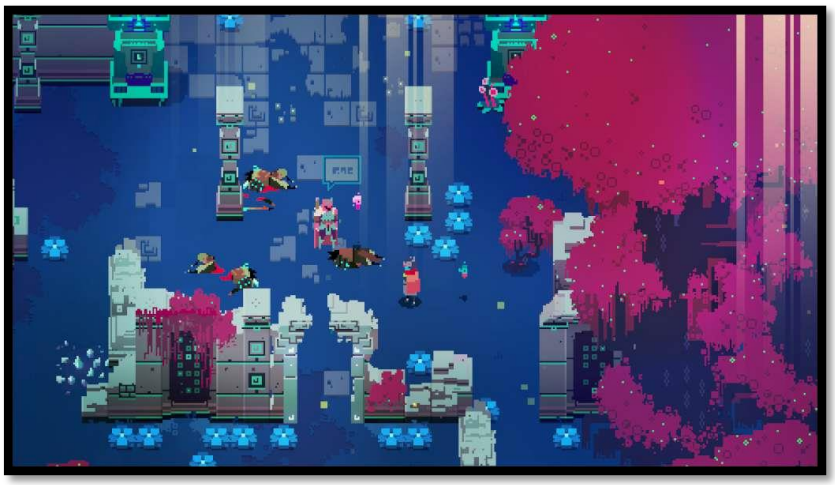


Figura 4 Hyper Light Drifter (2016)

Fuente: www.kinguin.net

JRPG

Los juegos JRPG (*Japanese Role Playing Game*) toman una aproximación más fiel a los juegos de rol de mesa tradicionales. En estos juegos, el protagonista suele contar con un equipo de personajes para asistirle durante la aventura. En estos juegos, los combates se suelen realizar por turnos, en los que el orden de los combatientes puede ser determinado por una estadística en particular, al azar o por una combinación de ambas. Algunos de los juegos más conocidos de este género son *Final Fantasy* y *Dragon Quest*, sagas que, aunque fueron de las primeras en aparecer en este mundo, aun hoy siguen lanzando títulos al mercado.

Figura 5 Final Fantasy IV (adaptación para Gane Boy Advance, 2005)

Fuente: www.lparchive.org

SRPG o TRPG

Si los juegos JRPG se acercaban mucho a los juegos de rol tradicionales, los SRPG (*Strategy Role Playing Game*) o TRPG (*Tactical Role Playing Game*) son una adaptación fiel casi al 100% de estos. En estos juegos, el jugador puede esperar encontrar una profundidad, tanto a la hora de combatir como de progresar por el mundo digna de los *tabletops* más famosos. Además, en estos juegos, la aleatoriedad juega un factor más importante, para simular los dados de veinte caras de los juegos tradicionales y puede, en ocasiones, significar la victoria o una humillante derrota para el jugador.

Estos juegos pueden dividirse en dos categorías, dependiendo de si son basados en texto o no. En el primer tipo encontramos juegos como el reciente *Blood Money* mientras que en el segundo encontramos otros como *Darkest Dungeon* o *Slay the Spire*

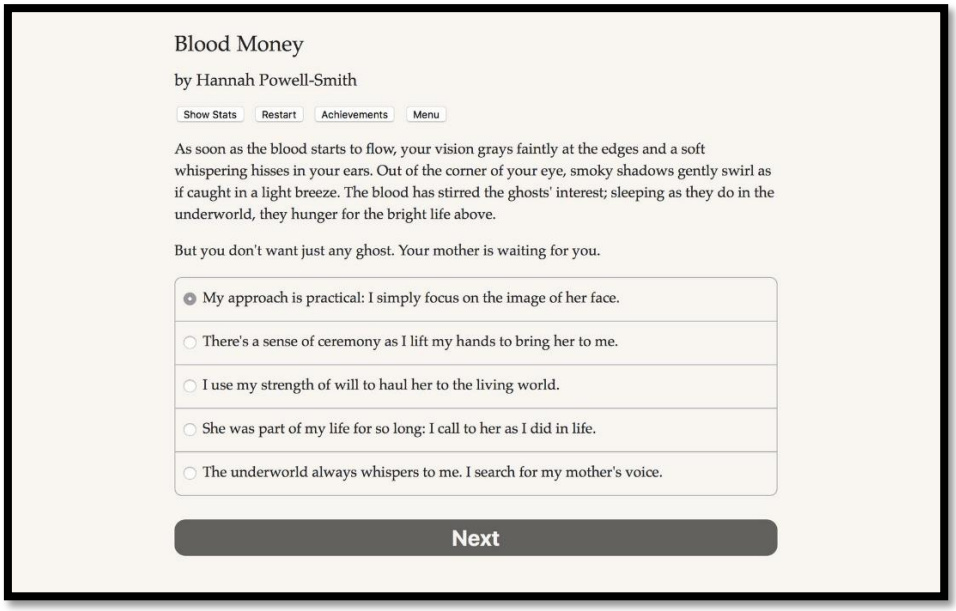


Figura 6 Blood Money (2018)

Fuente: www.whatsonsteam.com

Figura 7 Darkest Dungeon (2016)

Fuente: www.nuuvem.com

MMORPG

Siglas de *Massive Multiplayer Online Role Playing Game*. Estos son, normalmente, ARPG cuyos mundos y combates se orientan a la colaboración para poder exprimir el componente multijugador. En estos juegos, los acompañantes de los JRPG son reemplazados por otros jugadores. Títulos de este género pueden ser el archiconocido *Word of Warcraft* o *Guild Wars*.

Figura 8 World of Warcraft (2004)

Fuente: www.takemyscars.com

Juegos RPG en la actualidad

En los últimos años, los videojuegos RPG han tomado un viraje del formato JRPG (combate por turnos, movimiento en cuadrilla, compañeros NPC, etc.) hacía un formato más orientado a la acción, ARPG, en los que se pone a prueba al jugador con combates complicados y mecánicas extensas, las cuales tendrá que dominar para alcanzar la victoria. Estas mecánicas suelen estar enfocadas a la reactividad del jugador como movimientos de evasión, bloqueos con un escudo o ataques críticos en un punto débil del enemigo [4].

Además, muchos de estos juegos han dejado de lado la profunda historia que caracterizaba a los RPG de antaño para adoptar un estilo más arcade en el que el jugador tendrá que probar su valía y superarse una y otra vez para llegar cada vez más lejos. Normalmente, superar una fase del juego hace que la misma sea más sencilla en posteriores sesiones de juego. Juegos que enfrentan al jugador una y otra vez ante una mazmorra o desafío son conocidos como *dungeon crawlers* o *roguelite/roguelike* [5, 6].

Figura 9 Crawl (2014), un dungeon crawler

Fuente: crawl.wikia.com

Cabe destacar que no todos los juegos han abandonado la historia que caracteriza a los RPG, juegos como *The Witcher* o *Dark Souls* han tomado este enfoque de un ARPG con una densa y profunda historia y han sentado precedente, siendo dos de los mejores juegos valorados en este género.

Figura 10 Dark Souls (2009)

Fuente: www.sufficientvelocity.com

Debido al reinado actual de los ARPG en el mundo de los RPG y que personalmente considero que a un jugador se le debe presentar un reto y forzarlo a superarlo, el videojuego de este TFG será un ARPG *Dungeon Crawler* en dos dimensiones.

Motores de desarrollo de videojuegos

A la hora de desarrollar un videojuego hay dos formas de hacerlo:

- **Desde cero o *from scratch***: en las que el desarrollador usa librerías y utilidades para programar por completo el juego. Esto incluye funciones básicas como recoger las acciones del teclado o ratón (*input*), mostrar la ventana del juego y representar en ella el juego. El desarrollador podría dedicar tiempo a programar un editor que le ayude a diseñar niveles o crear y añadir características al juego.
- **Usando un motor de desarrollo de videojuegos o *game engine***: Con esta opción el desarrollador solo tiene que, en la mayoría de los casos, implementar el juego en sí. Este método tiene la ventaja de agilizar el trabajo, aunque suele arrebatar al desarrollador control sobre algunas funcionalidades como la gestión de memoria. Estos motores suelen traer consigo un editor listo para ser usado, lo cual agiliza aún más el proceso.

Las comodidades que ofrecen y el tiempo del que disponía me llevaron a decantarme por la segunda opción. A continuación, se hará un análisis comparativo entre los motores presentes en el mercado.

CryEngine

Figura 11 Imagotipo de CryEngine

Fuente: www.presseportal.de

CryEngine es un motor de juego desarrollado por la empresa alemana Crytek y, aunque comenzó siendo solamente un motor de demostración de Nvidia, acabó convirtiendose en un motor de desarrollo bastante popular desde que en 2004 saliera al mercado *Far Cry*, un juego del género FPS, desarrollado por la misma Crytek. Más tarde, en 2006, Ubisoft compraría la totalidad de los derechos de CryEngine [7].

CryEngine es capaz de producir juegos con un alto nivel visual gracias a la extensa gama de especificaciones técnicas que tiene. No obstante, su mayor problema de cara a este proyecto es que es un motor sumamente orientado al desarrollo de juegos 3D. El motor cuenta, además, con

una amplia documentación [8], una comunidad de tamaño moderado y una tienda de *assets* con multitud de productos.

Entre los juegos producidos en CryEngine encontramos algunos como la saga *Far Cry*, la saga *Crysis* o *Archeage* [9].

Xenko Game Engine

Figura 12 Imagotipo de Xenco Game Engine

Fuente: www.cgmagonline.com

Xenko es un motor de desarrollo de videojuegos orientado tanto a 3D como a 2D, creado por la empresa Silicon Studio y de código abierto u *open source*. Aunque la primera vez que se escuchó el nombre Xenko fue en 2014 cuando su repositorio se hizo público, aún tardaría varios años en abandonar la fase beta, en 2017 [10].

Aunque cuenta con varias características interesantes y necesarias en un motor de juego, la relativa juventud de Xenko hace que, aunque la documentación sea extensa y clara, la comunidad sea de un tamaño muy reducido y eso se traduce en menor volumen de recursos como tutoriales o demostraciones. Debido también al breve periodo de vida de Xenko, actualmente, solo un juego se encuentra en el mercado en fase de acceso anticipado: *Children of the Galaxy*, aunque muchos otros se encuentran en desarrollo [11].

Unreal Engine 4

Figura 13 Imagotipo de Unreal Engine

Fuente: ru.kisspng.com

Uno de los motores de juegos más conocidos y ya establecido en el mercado de estos. Desarrollado por Epic Games, Unreal Engine ha sido premiado en numerosas ocasiones, incluyendo premios como mejor motor de desarrollo y mejor aspecto visual [12]. Aunque se pueda argumentar que es el mejor motor del mercado en cuanto a 3D, se queda corto tanto en características como en optimizaciones para el desarrollo 2D, por lo que fue descartado para este proyecto.

Cuenta también con una gran documentación, masiva comunidad y un mercado de *assets* repleto de productos de todo tipo. Entre los juegos realizados en Unreal Engine encontramos grandes títulos como la saga *Bioshock*, *Borderlands*, *Unreal Tournament* o *Fortnite* [13]. Estos últimos dos siendo desarrollados por la misma Epic Games.

Godot Engine

Figura 14 Imagotipo de Godot Game Engine

Fuente: www.wikipedia.org

Godot es un motor de juego, orientado tanto a 2D como a 3D, de código abierto y desarrollado principalmente por la empresa argentina OKAM y por la propia comunidad de Godot. El código fuente fue liberado en 2014 y desde entonces ha recibido actualizaciones hasta llegar a Godot 3.0, que fue lanzado en abril del año 2017.

Godot, al igual que CryEngine, tiene una comunidad de tamaño moderado, una extensa documentación y varios recursos como tutoriales, además de una tienda de *assets*. Algunos de los juegos desarrollados en *Godot* son *Steam Quest* o *Deponia* [14].

RPG Maker: MV

Figura 15 Logotipo de RPG Maker MV

Fuente: www.masgamers.com

Como su propio nombre indica, RPG Maker es un motor orientado totalmente al desarrollo de juegos RPG, especialmente del género JRPG. Cuenta con una gran comunidad y con una riquísima tienda de *assets* que facilitan el trabajo de *scripting*. Sin embargo, como ya se ha mencionado en la sección *Juegos RPG en la actualidad*, el objetivo de este TFG es el de desarrollar un juego ARPG y no un JRPG, y, aunque es una tarea posible mediante el uso de *plugins* [15], el resultado dista del deseado para el proyecto.

RPG Maker, más que otros motores, sufre de un grave problema que perjudica a la popularidad de sus juegos: las diferencias entre juegos son mínimas. Esto no implica que no se pueda hacer un juego que sea un éxito en el mercado, como es el caso de *To the Moon* [16].

GameMaker Studio 2

Figura 16 Imagotipo de Game Maker Studio 2

Fuente: www.aprendegamemaker.com

GameMaker Studio 2 es un motor de juego desarollado por YoYo Games y el sucesor de GameMaker Studio, el motor con el que empecé a crear prototipos. Está extremadamente

orientado a juegos 2D, aunque también se puede crear un entorno 3D. Lamentablemente, dicha tarea es enormemente tediosa debido a las limitaciones tanto existentes en el motor en si como el hecho de que sea un motor principalmente 2D. Además, los resultados quedan muy por detrás en cuanto a calidad se refiere en comparación con otros motores [17].

Una de las razones por las que decidí dejar de usar GameMaker Studio fue que el sistema de interfaz es muy pobre en comparación con probablemente todos los demás, teniendo que programar desde cero tanto que la interfaz se mantenga en pantalla como que, por ejemplo, una *nine-slice box* funcione como tal. Además, el editor de habitaciones, que es el nombre que GameMaker da a las escenas, es escueto y dificulta la creación de niveles principalmente por no otorgarte una jerarquía visible. Con el lanzamiento de GameMaker Studio 2, muchos problemas de la anterior versión se solucionan, aunque los principales, siendo estos un sistema de interfaz inexistente y un editor pobre, siguen presentes. Para empeorar las cosas, la versión gratuita de GameMaker Studio 2 limita los juegos a las siguientes cantidades de elementos [18]:

Objetos	15
Sprites	20
Sonidos	10
Tilesets	2
Scripts	10
Paths	5
Timelines	2
Fuentes	5
Habitaciones	5
Shaders	0
Ficheros añadidos	0
Extensiones	0
Configuraciones	0

Tabla 1 Limitaciones de la versión gratuita de GameMaker Studio 2

Estas limitaciones han hecho que la comunidad de GameMaker haya mermado o haya preferido quedarse en la anterior versión donde la versión gratuita no era tan limitada. Esto causa, además, que los recursos como tutoriales o scripts tienen que seguir existiendo para ambas versiones y que está en manos del usuario decidir cuales buscar, atendiendo a la que ellos usen. Esto se aplica también a aquellos materiales presentes en el *asset store*.

Aún con todos los aspectos negativos de GameMaker, es un motor potente y permite crear rápidamente prototipos y juegos sencillos. Aunque han llevado largo tiempo de desarrollo, otros juegos que han sido éxitos de mercado como *Undertale* o el ya mostrado anteriormente en la sección *ARPG* *Hyper Light Drifter* han sido creados en GameMaker [19].

Unity 2018

Figura 17 Imagotipo de Unity

Fuente: www.evensi.com

Unity Engine es un motor de juego desarrollado por Unity Technologies, que surgió en 2005 como un motor exclusivo para Mac. El éxito que tuvo permitió a sus desarrolladores extender sus funcionalidades hasta llegar al entorno multiplataforma que es hoy. En sus principios, fue, también, exclusivamente dedicado al 3D, aunque, versión tras versión ha ido incluyendo funcionalidades 2D que han hecho que se convierta en uno de los principales motores para el desarrollo en dos dimensiones junto con GameMaker.

Unity forma, junto con Unreal Engine, la pareja de los principales motores de desarrollo de videojuegos, y, aunque años atrás, Unreal era claramente el mejor motor, Unity ha ido mejorando tanto sus características como sus números, superando a Unreal como el motor de desarrollo más usado en el año 2016 [20]. Muestra de esto es como cada vez estudios AAA como Blizzard desarrollan videojuegos enteramente usando este motor, como es el caso de *Hearthstone: Heroes of Warcraft* y como cada vez, más estudios hacen el cambio a Unity, siendo TellTale Games un ejemplo de ello [21].

Unity cuenta, además, con una gran comunidad debido al hecho de ser gratis y de tener una curva de aprendizaje media, que permite encontrar recursos como tutoriales, scripts o respuestas a problemas que se le puedan presentar al usuario. El tamaño de la comunidad también ayuda al número de *assets* presentes en la tienda, lo que facilita la tarea de empezar o terminar un proyecto.

Los títulos realizados en Unity incluyen nombres de juegos que han sido calificados como referentes en sus géneros como *Hollow Knight* o *Cuphead* [22].

Conclusiones

A continuación, se muestra una comparación de los motores vistos anteriormente atendiendo a varias de sus características como el lenguaje de programación, la arquitectura o la curva de aprendizaje:

	CRYENGINE	**XENCO**	**UNREAL ENGINE**	**GODOT**	**RPG MAKER MV**	**GAME MAKER STUDIO 2**	**UNITY**
Lenguaje de programación	C++, C#	C#	C++, blueprints	GDSCript (similar a Python), C#, C++, blueprints	RGSS (Ruby Game Scripting System)	GML (similar a Java), Drag and Drop (DnD)	C#, UnityScript (en desuso), BOO
Arquitectura	POO	POO	Componentes	POO	POO	Componentes	Componentes, ECS (Entity Component System)
Orientación	3D	3D	3D, 2D	3D, 2D	2D	2D	3D, 2D
Curva de aprendizaje	Media	Media	Alta	Media	Baja	Baja	Media
Asset store	Sí	Sí	Sí	Sí	Sí	Sí	Sí
Gratuito	Sí	Sí	Sí	Sí	Sí	No	Sí
*Tamaño de la comunidad**	1219 [23]	No disponible	36487 [24]	10474 [25]	9784 [26]	26243 [27]	87086 [28] para 3D, 35735 [29] para 2D

Tabla 2 Comparativa entre motores de juego

** Tomado como referencia el tamaño de las comunidades de Reddit.*

Aunque apreciamos que todos los motores son fundamentalmente bastante similares, a lo largo de todo este apartado hemos podido concluir tres cosas:

1. Solo Unity, Godot, GameMaker Studio, RPG Maker están realmente capacitados para el desarrollo en 2D.
2. Solo Unity, Godot y GameMaker Studio permiten desarrollar un buen sistema de combate de acción, fundamental en un ARPG.
3. Solo Unity y Godot tienen un sistema de interfaz robusto y editores potentes.

Visto esto, concluimos que Unity y Godot son los más indicados para este proyecto, pese a ello, Unity tiene características que Godot no, como, por ejemplo, *timelines* para la composición de animaciones más complejas de más de un objeto. Además, Unity recibe actualizaciones constantes al ser el producto principal de una empresa y tiene una comunidad mucho mayor, que puede ayudarme durante el desarrollo del videojuego y puede hacer que la difusión del juego llegue a más gente. Con una ajustada victoria, Unity se postula como el mejor motor para este proyecto.

Referencias e inspiración

En este apartado se mostrarán algunos de los juegos que han servido de inspiración para el diseño y el desarrollo del videojuego, analizando los aspectos positivos y negativos de cada uno con intención de aplicar o adaptar aquellos puntos fuertes y evitar los aspectos negativos de otros juegos del mercado.

The Legend of Zelda: The Minish Cap

Figura 18 The Legend of Zelda: The Minish Cap

Fuente: www.amazon.com

La saga *The Legend of Zelda* siempre ha sido un referente en el género de RPG y aventuras. Uno de sus títulos: *The Legend of Zelda: Ocarina of Time* para la consola Nintendo 64 es considerado por muchos como el mejor juego de la historia [30]. Dichas listas suelen estar realizadas por revistas especializadas, y, aquellas que no consideran a *Ocarina of Time* como el mejor juego, lo hacen con *The Legend of Zelda: Breath of the Wild* [31, 32].

Aunque los triunfos de la saga no se limitan a aquellos títulos en 3D. Para la consola portátil Game Boy Advance fueron lanzados dos títulos: una remasterización del clásico *A link to the Past* y *The Minish Cap*, ambos juegos figuran en todas las listas de los mejores de juegos de Game Boy Advance solo por detrás de titanes como Super Mario o Pokémon [33, 34].
De entre ellos dos, he decidido basar mi inspiración principalmente en *The Minish Cap*, debido a lo bien diseñados que están sus niveles y como combinan perfectamente la interacción entre el mundo de tamaño normal y el mundo de tamaño microscópico.

Legend of Dungeon

Figura 19 Legend of Dungeon

Fuente: www.robotloveskitty.com

Legend of Dungeon es un *dungeon crawler* con un sencillo objetivo: superar todos los peligros hasta alcanzar el piso veinticinco, recuperar el tesoro y volver sano y salvo a la superficie. Cada nivel del calabozo se compone de diferentes habitaciones, totalmente diferentes y aleatorias, con trampas, peligros y enemigos de todo tipo. Además, los objetos son también generados de manera procedimental, dando así lugar a que no existan dos objetos iguales.

Con una premisa y un combate sencillo, *Legend of Dungeon* se postula como un fantástico *roguelite*, siendo su punto fuerte la fantástica generación procedimental del entorno. Sin embargo, el manejo del inventario es tedioso, ya que no está limitado y el jugador podría tener tantos objetos como haya encontrado, de los cuales solo podría ver tres al mismo tiempo.

Heroes of Hammerwatch

Figura 20 Heroes of Hammerwatch

Fuente www.heroesofhammerwatch.com

Heroes of Hammerwatch es el sucesor de *Hammerwatch* desarrollado por Crackshell. En estos juegos, el objetivo es llegar al último nivel y derrotar al dragón que está aterrorizando la ciudad. Esta tarea no es fácil, ya que el jugador deberá gestionar los recursos encontrados en la mazmorra para poder mejorar el equipamiento de su personaje y así facilitar las siguientes aventuras. Por eso, un jugador que empiece a jugar, a duras penas llegará al cuarto nivel, mientras que un jugador que haya jugado más tiempo y haya conseguido más recursos, podrá superar esos niveles fácilmente.

Aun así, no es este aspecto de la economía el punto fuerte de *Heroes of Hammerwatch*, si no su combate. En *Heroes of Hammerwatch*, los niveles están repletos de enemigos, pudiendo contener varios cientos de ellos en un solo nivel. Aunque suene intimidante, el sistema de combate orientado a habilidades con diferentes niveles de poder hace que sea posible derrotar a una gran cantidad de enemigos. Además, esto influye en la sensación que el juego transmite: llegar a un piso lleno de enemigos hace que el jugador se estremezca antes los peligros que rondan en el calabozo, sensación que contrasta con la sensación de poder que transmite lanzar un hechizo poderoso que elimina a varias decenas de enemigos.

Diablo II

Figura 21 Diablo II

Fuente: www.pu.nl

El juego ARPG por excelencia y que marcó las infancias de muchos niños que crecieron jugando a videojuegos. Aún hoy se le considera un referente en el género, más aún que a *Diablo III*, su sucesor.

En *Diablo II* no solo las mazmorras son aleatorias, si no que el mundo, y todo aquello que lo habita (a excepción de enemigos básicos y zonas de jefes finales o ciudades, que son pseudoaleatorias) está generado de manera procedimental, desde objetos hasta mini jefes. Cuenta además con un fantástico sistema de combate orientado a habilidades, que, al igual que *Heroes of Hammerwatch*, tienen diferentes niveles de poder. No obstante, los objetos pueden tener características que mejoren o amplifiquen tus habilidades, haciendo que el estilo de juego vaya cambiando conforme el jugador progresa.

Finalmente, el sistema de inventario de la saga Diablo, en el que el espacio del inventario está dividido en una matriz y los objetos ocupan espacios rectangulares de diferentes tamaños como 2x2, 1x3 o 3x4, es excelente y ha sido implementado en multitud de juegos a lo largo de la historia.

Animes del género Isekai

Figura 22 Log Horizon

Fuente: myanimelist.com

La palabra japonesa *isekai* (異世界) significa "mundo alternativo". En los animes pertenecientes a este género, el protagonista es trasportado a otro mundo en el que debe sobrevivir o cumplir un propósito. En los últimos años han sido publicados varias series de este género en el que el protagonista aparece en el mundo de un juego RPG, como es el caso de *Log Horizon* o *Kono Subarashii Sekai Ni Shukufuku O!*, comúnmente abreviado como *Konosuba*.

Aunque no son videojuegos, adaptan ese aspecto a la historia y se centran en el desarrollo de personajes, hay un matiz que estos animes comparten y que es muy complicado de llevar a un videojuego sin que resulte frustrante: Cualquier enemigo es una amenaza mayor. Por ejemplo, es típico jugar a un RPG y que el primer enemigo que nos encontremos sea una babosa, ya que siempre se ha tratado a estos enemigos como los más débiles en estos juegos.

Figura 23 Una babosa en Dragon Quest

Fuente: www.dragon-quest.org

Ocurre al contrario en *Konosuba*, donde la única babosa que aparece en los veinte episodios es un peligroso monstruo toxico capaz de contaminar las aguas de una región y derretir a sus contrincantes.

Figura 24 Una babosa en Konosuba

Fuente: konosuba.wikia.com

Es esta sensación de peligro y miedo ante cualquier enemigo que tan bien transmiten los personajes de las series de estos géneros la que yo quiero transmitir con mi juego.

Principales características del juego

Ahora, se detallarán los aspectos clave del videojuego y se comparará como otros juegos han implementado esas características. Para disponer de una base antes de empezar el desarrollo.

Generación procedimental de mazmorras

La generación procedimental de entornos y elementos tiene varios aspectos a favor y varios en contra [35, p. 13-16]:

- Beneficios
    1. **Permite la creación de juegos de mayor escala.** En un videojuego, todo aquello que aparece en el mundo debe ser creado y colocado a mano. Si un juego utiliza la generación procedimental de manera correcta, teóricamente, no existiría un límite para el tamaño del mundo. Ejemplo de esto es *No Man's Sky*, un juego de ciencia ficción situado en una galaxia infinita generada procedimentalmente. Pese a esto, existe un factor limitador en el hardware, ya que mundos de tamaño ingente necesitarían muchísima memoria: solo para almacenar la información del mundo más grande posible de *Minecraft* se necesitarían 409 *petabytes* de almacenamiento.
    2. **Reduce costes de producción.** Pongamos, como ejemplo, un mundo que necesite cien texturas de hierba diferentes. El coste de crear esas cien texturas manualmente es mucho mayor que el de crear cinco y programar un sistema que las combine y genere las otras noventa y cinco de manera procedimental. Esto se aplica también al modelado, sonido y especialmente al diseño del juego o *game design*.
    3. **Aumenta la variedad dentro del juego.** En un mundo hecho a mano, todos los jugadores, al final, tendrán los mismos objetos, habrán luchado contra los mismos enemigos y habrán vivido, en definitiva, las mismas aventuras y experiencias. Por el contrario, la característica principal de algo generado procedimentalmente es que las experiencias difieren.
    4. **Aumenta la rejugabilidad.** Continuando con el anterior punto, la experiencia de recorrer un mundo procedimental varía, no solo entre distintos jugadores, sino también entre distintas sesiones de un mismo jugador.
- Inconvenientes
    1. **Más exigente para el hardware.** La generación procedimental se basa en la ejecución de algoritmos. Estos algoritmos suelen ser muy pesados y hay que asegurarse de que el sistema del jugador medio pueda soportarlo.
    2. **Los mundos pueden ser repetitivos.** Cuando se desarrolla un algoritmo de generación pobre y poco parametrizado, es común que se perciban patrones y

repeticiones. Además, si el mundo es demasiado grande, se acabarán agotando las diferentes posibilidades y alguna parte de este puede salir igual a otra.

3. **Sacrifican control de calidad.** Aunque los ordenadores son extremadamente buenos resolviendo algoritmos, siempre serán inferiores a los humanos en cuanto a creatividad. Al dejar que el sistema genere el mundo, se está perdiendo el toque humano que un buen diseñador aporta a un juego. Otro inconveniente de esto es que tampoco se garantiza la calidad de *gameplay*. A un jugador puede tocar un mundo muy fácil con objetos poderosos que le permita pasarse el juego fácilmente mientras que a otro le toque justo lo contrario.
4. **Existe la posibilidad de generar un mundo injugable.** Si el algoritmo no es lo suficientemente robusto, se podrían generar terrenos demasiado altos que el jugador no pueda escalar o puertas cerradas sin una llave accesible. Esto se debe a la casi nula certeza que proporcionan estos tipos de algoritmos.
5. **Es difícil crear eventos definidos.** Como continuación del punto anterior, si, por ejemplo, se quiere que ocurra una emboscada en la habitación central del nivel cinco de una mazmorra, no se puede tener la certeza de que dicha habitación existirá, o de que, si lo hace, sea del tamaño adecuado para que la emboscada sea jugable. Por definición, eventos predefinidos son incompatibles con un algoritmo que genera mundo no predefinidos. La solución a este problema es forzar el evento dentro del algoritmo: si nos encontramos en el nivel cinco, forzar al algoritmo a generar una habitación de un tamaño predefinido con un disparador que dé lugar a la emboscada. Esto resta aleatoriedad y variedad a nuestro juego, ya que el jugador puede esperar que la habitación de la izquierda del nivel cinco sea de tamaño 7x7, por ejemplo, y que al entrar en ella tendrá que sobrevivir a una emboscada.

Habiendo visto la teoría sobre la generación procedimental, podemos ahora observar como algunos juegos del mercado se enfrentan a los problemas presentados en la lista de desventajas y como aprovechan los beneficios de estos algoritmos.

Nuclear Throne

Figura 25 Nuclear Throne

Fuente: www.construct.net

Los niveles en *Nuclear Throne* y juegos similares como *Dungeon Souls* son "esculpidos" partiendo de una matriz de paredes. Se siguen los siguientes pasos:

1. Se crea una matriz y se definen las dimensiones tanto de la propia matriz, como de las celdas que la componen. El tamaño de las celdas será el mismo que el tamaño de los *tiles* que usemos.
2. Se crea un objeto que se mueve de forma aleatoria por la matriz. Las casillas en las que se sitúe se etiquetan como suelo. Esto ocurrirá un número predeterminado de veces.
3. Una vez se tienen diferenciados las paredes de los suelos, se colocan los *tiles* atendiendo si la casilla en la que se sitúan está etiquetada como suelo o como pared, y en este último caso, habrá que fijarse también en las paredes vecinas para colocar el *tile* correspondiente.

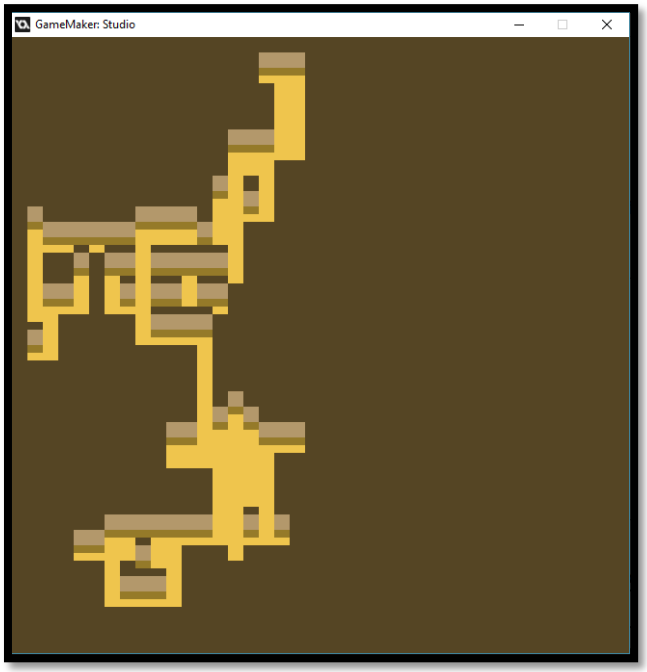


Figura 26 Ejemplo de generación de mapas al estilo Nuclear Throne implementada por mí en GameMaker(2016)

Fuente: elaboración propia

Figura 27 Tileset utilizado para el ejemplo anterior (las zonas blancas son transparentes)

Fuente: elaboración propia

4. Una vez llegados a este punto, el nivel está casi terminado. Lo único que quedaría sería definir un punto de partida, colocar al jugador en él, definir un punto de salida para avanzar al siguiente nivel, añadir enemigos, trampas, objetos y dar un poco de vida al nivel con elementos como barriles, cajas, hierbas o cambios en el patrón del suelo.

Uno de los aspectos positivos de este tipo de algoritmos es su facilidad a la hora de implementarlo mientras que el poco control que se tiene sobre el resultado juega en su contra: podría aparecer un nivel con equilibrio entre pasillos y habitaciones o podría aparecer una habitación gigante. Podría, también, no haber enemigos o haber uno en cada casilla posible. Aunque todo esto depende de cómo de parametrizado esté el algoritmo, siempre existirá esa incertidumbre respecto al resultado y el que quizá sea el mayor de los problemas de este tipo de algoritmos: el nivel no tiene nada de humano y no tiene coherencia aparente.

The Binding of Isaac

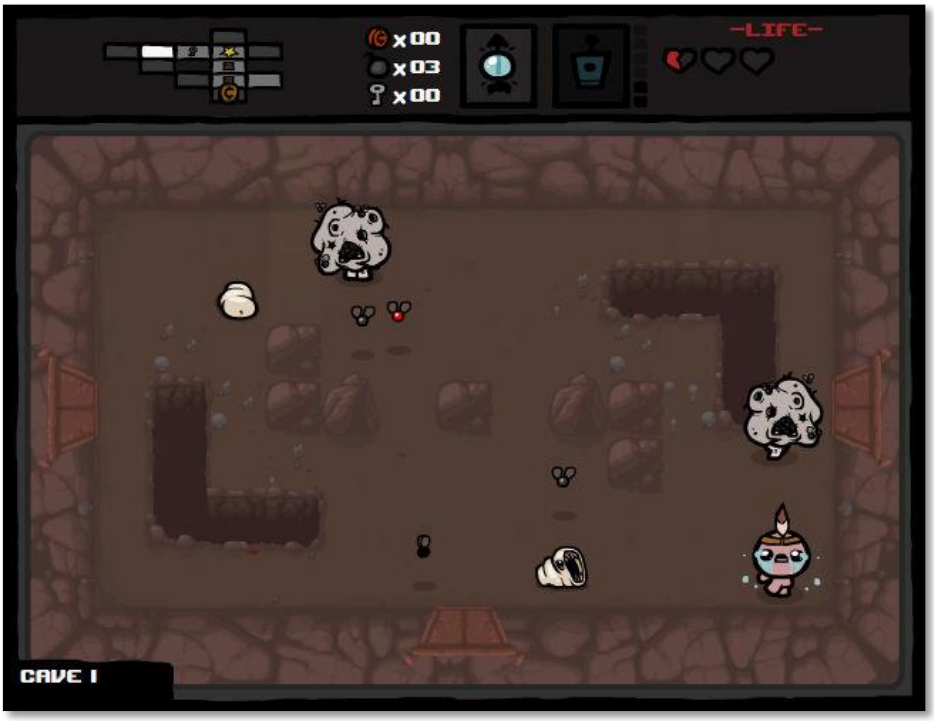


Figura 28 The Binding of Isaac

Fuente: dualshockers.com

En *The Binding of Isaac*, la matriz de elementos no genera el nivel en sí como en *Nuclear Throne*, si no que los elementos son habitaciones en las que se desarrolla el juego. Estas habitaciones se crean desde plantillas diseñadas a mano en las que se define qué objetos del juego aparecerán en cada una de las casillas de la habitación, ya que se puede apreciar que la habitación en sí es una matriz.

Figura 29 Demostración de la naturaleza de matriz de una habitación en The Binding of Isaac

Fuente: elaboración propia a partir de la figura 28

Símbolo	**Objeto correspondiente**
Letras	Columnas
Números	Filas
Líneas blancas	Limitadores de celdas
Círculos verdes	Localización de puerta

Tabla 3 Leyenda de la figura 29

Basándonos en la figura 29 y la leyenda proporcionada, apreciamos que las habitaciones son una matriz de 7x13. La razón por la cual se usan números impares es para que las puertas queden siempre en el centro. Además, podemos ver en el mini mapa, arriba a la izquierda, que la habitación actual (de color blanco), no tiene una habitación vecina arriba, mientras que, si las tiene hacia la derecha, abajo e izquierda, siendo estas direcciones en las que tiene puertas. Esto explica el hecho de que, si bien una habitación no siempre tendrá una vecina en alguna dirección, en caso de tenerla siempre estarán conectadas mediante una puerta.

Vemos también que el diseñador, a la hora de realizar la plantilla, ha colocado diversos elementos formando patrones. Esto le da el toque humano y la cohesión que faltaba en los niveles de *Nuclear Throne*. Además, esto permite fácilmente añadir elementos diferentes: por

ejemplo, la casilla 3, D es una roca, que impide al jugador y enemigos moverse a través de ella y disparar, mientras que la casilla 4, B es un agujero, que no impide disparar a través de ella y que solo los enemigos voladores podrán atravesar.

Conclusiones

Habiendo visto esto, podemos deducir que, aunque los dos métodos de generación son influenciados por la aleatoriedad de una u otra forma, la aproximación de *The Binding of Isaac* permite que los niveles tengan un toque más humano debido a que, aunque el nivel a gran escala es procedimental, a escala de juego es un nivel diseñado a mano. Por estas razones, este es el método que se usará en el juego.

Sistema de equipamiento

El equipamiento son los objetos como armas, armadura o pociones que pueda poseer el jugador. Normalmente, estos objetos se almacenan en un inventario. El sistema de equipamiento es una parte fundamental en los RPG ya que transmite sensación de progreso y evolución.

Al igual que el sistema de inventario se puede implementar de muchas formas, el de equipamiento ofrece también múltiples posibilidades, cada una con sus ventajas y desventajas.

Sistema de equipamiento en Heroes of Hammerwatch

En *Heroes of Hammerwatch* no importa cuantos objetos posea el jugador, al aspecto de su personaje siempre será el mismo.

Figura 30 Un sacerdote en Heroes of Hammerwatch antes y después de comprar objetos

Fuente: elaboración propia.

Símbolo	**Objeto representado**
Circulo amarillo	Personaje del jugador
Rectángulo verde	Dinero del jugador
Rectángulo azul	Daño de ataque del jugador

Tabla 4 Leyenda de la figura 30

En la figura 30 se nos presenta un personaje de la clase sacerdote antes y después de comprar objetos en una tienda. Guiándonos con la leyenda (Tabla 4), podemos ver que el jugador dispone de 17697 monedas de oro, tiene un daño de ataque de 90 y va vestido con una túnica azul y una mitra también azul. Poco después, compra dos objetos de la tienda, quedándose con 14517 monedas de oro. Uno de estos objetos es *Markham's mace* o *Maza de Markham*, la cual otorga un punto de daño por cada objeto en el inventario, y, al tener dos objetos, el jugador goza de una bonificación de dos puntos de ataque, teniendo ahora 92. Pese a esto, apreciamos que el jugador sigue llevando su túnica y su mitra azules.

Este sistema tiene varias ventajas y desventajas. La principal ventaja es que el tiempo y dinero que se invierte en crear *sprites* es mínimo, ya que un personaje jugable tiene un bajo número de *sprites.*

Animación	*Frames*	¿Depende de la dirección?
Idle o estar quieto	1	Sí
Correr	2	Sí
Atacar	2	Sí
Morir	4	No

Tabla 5 Análisis de las animaciones en Heroes of Hammerwatch

Si sumamos la cantidad de *frames* que dependen de la dirección a la que el jugador mire, multiplicamos el resultado por el número de direcciones posibles (8 direcciones) y sumamos el número de sprites independientes de la dirección, obtenemos que un personaje jugable tendrá:

$$Número\ de\ sprites\ diferentes = (1 + 2 + 2) * 8 + 4 = 5 * 8 + 4 = 40 + 4 = 44\ sprites$$

Un personaje tendrá 44 *sprites* diferentes lo cual es una cantidad de trabajo mucho inferior a otros juegos del mercado.

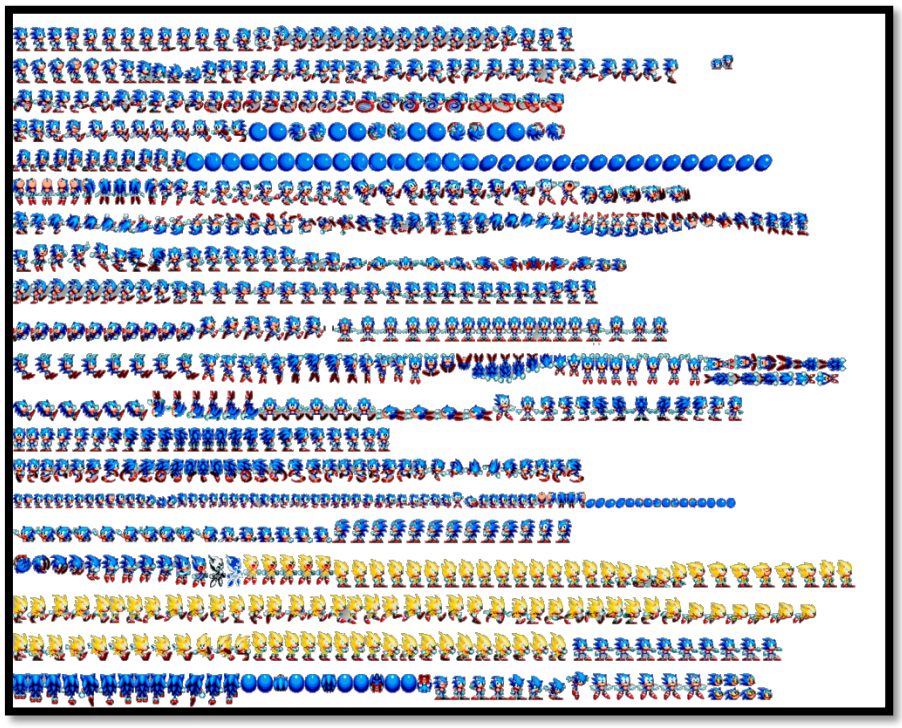


Figura 31 Spritesheet de Sonic en Sonic Mania, con alrededor de 600 sprites

Fuente: www.deviantart.com, por el usuario MarioGamer9000

Por otra parte, la principal desventaja de este sistema es la invariabilidad del personaje: no importa cuánto avance, tu personaje siempre se verá igual. Esto resta sentimiento de progreso al juego: aunque los objetos que el jugador recoja lo fortalezcan, el aspecto, que es lo que el jugador percibe principalmente va a ser el mismo.

Sin embargo, *Heroes of Hammerwatch*, el cual es un juego de naturaleza caótica, con mecánicas extraídas de los juegos *bullet hell* necesita la máxima claridad a la hora de jugar y ha sabido convertir esta desventaja en una ventaja.

Aun así, el juego en el que se centra este TFG no va a tener la misma cantidad de elementos en pantalla que *Heroes of Hammerwatch* por lo que no se podría convertir esta desventaja en una ventaja y la reducción de tiempo de este método no compensaría con la sensación de progresión perdida.

Sistema de equipamiento en The Legend of Zelda

En los juegos de *The Legend of Zelda*, tanto en 2D como 3D, el equipamiento está oculto hasta el momento de su utilización.

Figura 32 Spritesheet de Link, protagonista de The Legend of Zelda: A Link to the Past

Fuente: www.spriters-resource.com, por el usuario Mister Man

Aunque las animaciones del personaje y los objetos están divididas para reutilizar al máximo las animaciones, hay algunas que igualmente han tenido que ser hechas expresamente para un objeto, como la animación de disparar con arco.

Este método se sitúa como la alternativa neutral a otros métodos, ya que ahorra más tiempo que otros en cuanto al desarrollo de la *spritesheet*, pero no tanto como, por ejemplo, *Heroes of*

Hammerwatch. Ocurre de forma similar con el sentimiento de progreso. Se nota con objetos que están siempre visibles como el traje que cambia de verde a azul y finalmente a rojo o con el escudo, que pasa de azul a rojo y finalmente a amarillo. Sin embargo, no ocurre así con objetos como el *boomerang* o la espada, los cuales solo vemos al usarlo.

Figura 33 No podemos saber que espada está usando Link

Fuente: strategywiki.org

Figura 34 Link ya ha atacado, podemos ver que está usando la espada básica

Fuente: strategywiki.org

Llegados a este punto, hay que analizar si la sensación de progreso del juego es la deseada o al menos suficiente con este método, para así poder aprovechar el ahorro de tiempo.
En el caso del juego de este TFG, aun es insuficiente y no transmite el sentimiento deseado.

Sistema de equipamiento en juegos como Diablo II o MMORPG

En juegos como en *Diablo II* o un MMORPG, la armadura y las armas están siempre visibles, esto otorga una sensación de progreso insuperable, ya que el jugador pasa todo el tiempo mirando a su personaje con, por ejemplo, un yelmo determinado. Cuando días después consigue un yelmo mejor y que, además, tiene un aspecto más fiero o brillante, el jugador siente que su personaje es más fuerte solo con mirarlo y compararlo al anterior.

Normalmente, los MMORPG son 3D, lo cual facilita el desarrollo de este método de gestión de equipamiento. Dicha facilidad no existe en los juegos en 2D.

- Para implementar un método así en un juego 2D, hay que utilizar un número de capas completamente separadas equivalente al número de elementos variables del personaje como puede ser el casco, la armadura del pecho o la espada.
- En cada capa se muestra un *Sprite* perteneciente a su propia *spritesheet*.
- Todas las *spritesheets* tienen que encajar perfectamente entre ellas. En caso de no ser así, se producirían problemas de visualización.
- Las capas tienen que reordenarse dinámicamente en el eje Z (en principio, inexistente en el entorno 2D), para que la visualización sea coherente: si por ejemplo un personaje tiene un escudo en la mano derecha, este debería verse detrás del personaje cuando mire hacia la izquierda, y delante del jugador cuando mire a la derecha.

A continuación de hace una demostración del ensamblado de una spritesheet en capas.

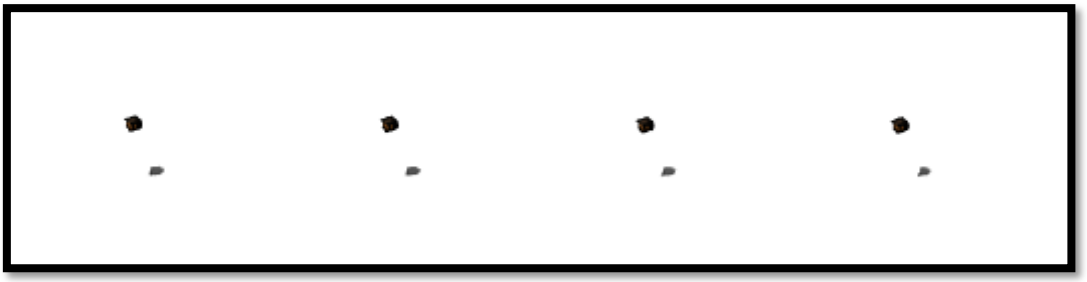


Figura 35 Cuatro frames de la spritesheet de la cabeza de "Isometric hero"

Fuente: www.opengameart.com, por el usuario Clint Bellager

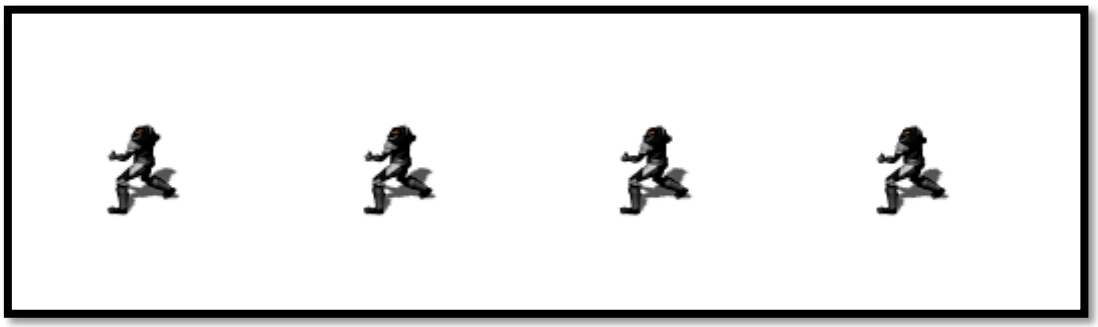


Figura 36 Cuatro frames de la spritesheet de la armadura de "Isometric hero"

Fuente: www.opengameart.com, por el usuario Clint Bellager

Figura 37 Cuatro frames de la spritesheet de la espada de "Isometric hero"

Fuente: www.opengameart.com, por el usuario Clint Bellager

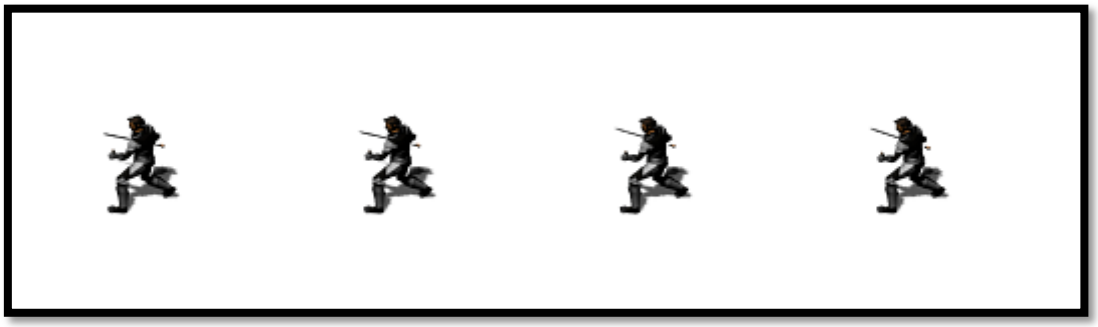


Figura 38 Cuatro frames resultante de la combinación de las anteriores spritesheets.

Fuente: www.opengameart.com, por el usuario Clint Bellager

Como podemos comprobar, al aspecto es bueno y nos permite mucha versatilidad a la hora de añadir nuevos objetos, por ejemplo, un bastón mágico.

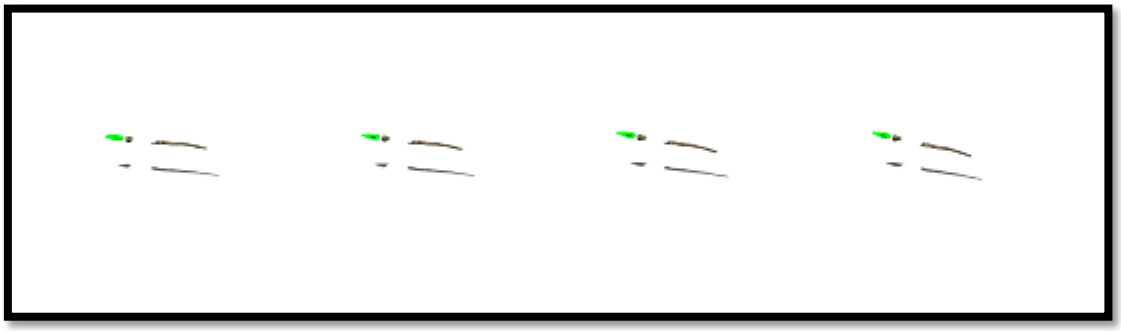


Figura 39 Cuatro frames de la spritesheet del bastón de "Isometric hero"

Fuente: www.opengameart.com, por el usuario Clint Bellager

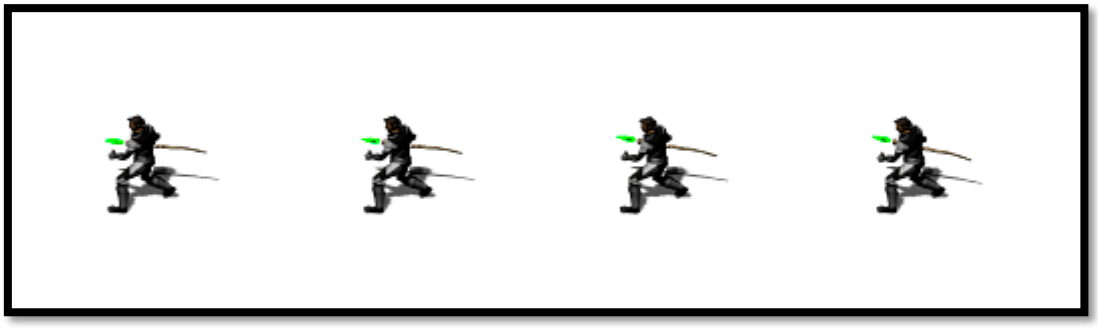


Figura 40 Resultado de combinar las spritesheets de la cabeza y la armadura con la del bastón

Fuente: www.opengameart.com, por el usuario Clint Bellager

Podemos también ver que, en este ejemplo, Clint Bellager ha optado por la técnica de la oclusión en las spritesheets: cuando un objeto o parte de él es tapado por otra parte del cuerpo, se borra la parte ocluida de manera manual de la spritesheet. Esto elimina la necesidad de reordenar las capas de manera dinámica, aunque, por otra parte, nos resta libertad a la hora de diseñar objetos, ya que, si existiese una armadura que, por ejemplo, tuviese una capa o unas hombreras muy grandes, ocasionaría problemas de visualización o nos obligaría a hacer una *spritesheet* aparte para todos los objetos. Una para cuando se usen en conjunto con la armadura sin capa y otra para cuando se usan con la armadura con capa.

La otra técnica consiste en que las *spritesheets* son compuestas por *sprites* al natural, es decir, tal y como son. Luego, al reordenar las capas, las partes ocluidas no serán visibles de manera natural. Aun así, este método tiene inconvenientes que yacen en el hecho de que no está preparado para los cambios de plano. Si por ejemplo quisiésemos que el escudo de la mano derecha se viese a la izquierda del personaje, seguramente tengamos que ocluir manualmente alguna parte del *sprite*. Aunque esto nos lleva de vuelta al mismo problema del método de las oclusiones, esta vez se reduce a unos pocos *sprites* y su impacto está más atenuado.

Viendo esto, podemos deducir la ingente cantidad de trabajo que hay detrás de este método, ya que hay que hacer todas y cada de las *spritesheets* a mano, y, además, hay que asegurarse de que encajen perfectamente. Hay que tener también mucho cuidado con qué partes quedan ocluidas en los sprites, independiente de si vamos a ordenar las capas o no, y, en caso positivo, establecer el orden y programar el algoritmo.

Como se puede esperar, la ventaja de este método es la sensación de progresión del personaje que transmite al jugador y bien que presenta a este el equipamiento que lleva, lo que incrementa la inmersión del jugador.

Antes de decantarse por este método, hay que preguntarse si la cantidad de trabajo que supone merece la pena. Aunque las *spritesheets* de ejemplo de Clint Bellager se componen de 256 *sprites*, estos son en realidad *renders* de un modelo 3D animado, que, aunque también lleva mucho trabajo, elimina la necesidad de plantear las direcciones en las que mira el jugador y la perspectiva. Para el juego del TFG, yo haré las *spritesheets* a mano y con estilo PixelArt, por lo que reduciendo el número de *sprites* podemos llegar a una cantidad de trabajo razonable para obtener un buen aspecto gráfico y una buena sensación de inmersión.

Conclusiones

El principal atractivo que quiero que el juego tenga es que transmita la sensación de ser parte de un mundo peligroso, oscuro y con enemigos, pero que se pueden superar mejorando tanto mecánicamente como estadísticamente (a través de objetos). Por eso para el desarrollo del videojuego, me decantaré por el último método, equilibrando la cantidad de trabajo para poder alcanzar los objetivos del proyecto a tiempo.

Sistema de combate

Como ya se mencionó en la sección *Juegos RPG en la actualidad*, el objetivo de este TFG es desarrollar un ARPG, y un aspecto fundamental de este tipo de juegos es su sistema de combate. Al igual que la generación procedimental de niveles y el sistema de equipamiento, hay muchas posibilidades a la hora de diseñar un sistema de combate robusto.

Sistema de combate basado en habilidades

En este tipo de sistemas, el jugador dispone de un número limitado de habilidades con diferentes efectos como daño, potenciadores o área de efecto. Está en manos del jugador usar estas habilidades a su máximo potencial.

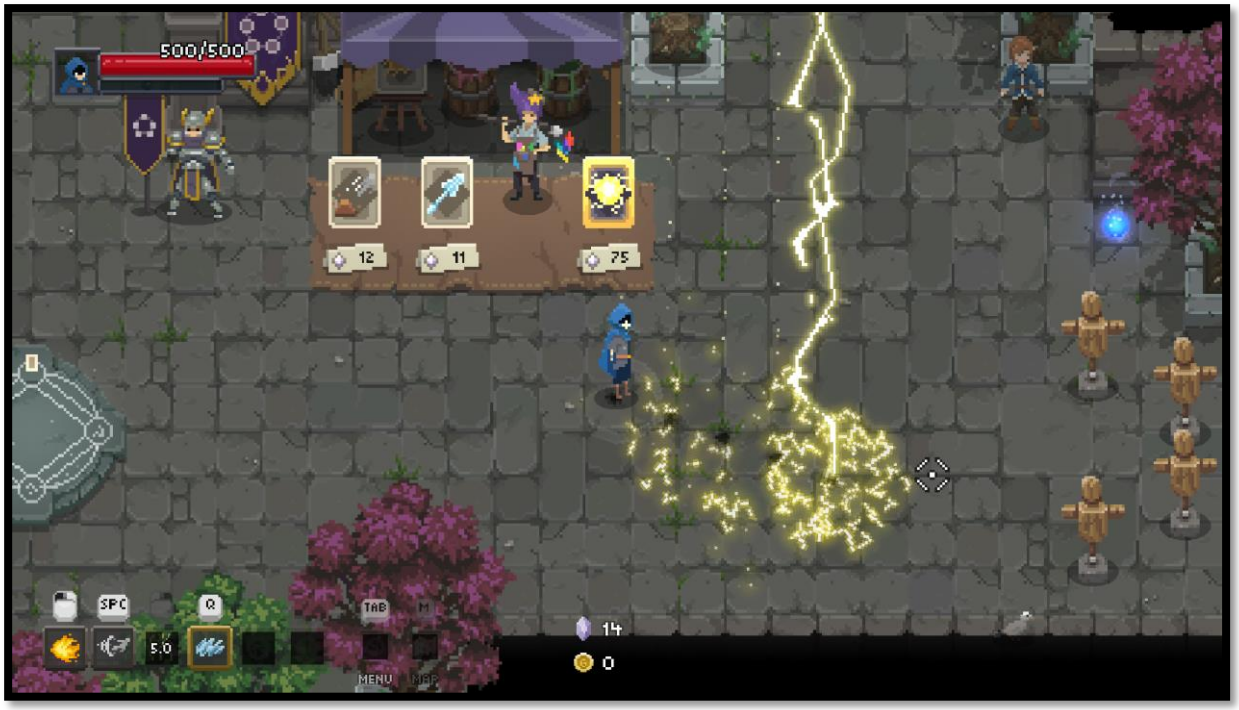


Figura 41 Wizard of Legend, un roguelite basado en habilidades

Fuente: elaboración propia

En la figura 41 se muestran, abajo a la izquierda, las habilidades de las que dispone el jugador. Las habilidades, por lo general, suelen tener restricciones como un coste de energía necesario para usarlas y un tiempo de enfriamiento el cual tendrá que terminar por completo para poder volver a usar la habilidad.

La ventaja principal de este sistema es que el jugador se siente poderoso al lanzar habilidades poderosas y con bellos efectos visuales. Sin embargo, el coste de producción del juego aumenta enormemente, ya que hay que diseñar suficientes habilidades para que el jugador pueda probar varias combinaciones o corremos el riesgo de que el juego sea aburrido.

Este sistema es, además, muy intensivo en cuanto a equilibrio se refiere. Una habilidad muy poderosa estará presente en todas las combinaciones que haga un jugador, limitando el número de posibilidades. Por otra parte, una combinación de habilidades muy poderosa dejará obsoletas al resto. Finalmente, encontramos en el otro lado del espectro, aquellas habilidades y combinaciones débiles o muy específicas, que serán normalmente ignoradas por el jugador y pueden llegar incluso a generar frustración al desbloquearlas si el jugador no tiene control sobre el proceso de desbloqueo.

Sistema de combate basado en combos

Este sistema es la siguiente iteración del anterior. Aunque con un juego con habilidades el jugador puede realizar combos con ellas, estos serán, en esencia, un placebo, ya que en realidad solo son una sucesión especifica de habilidades que el jugador ha decidido usar. Sin embargo, en un juego con un sistema basado en combos, las habilidades lanzadas en un orden determinado se convierten en habilidades totalmente diferentes en cuanto a efectos visuales, propiedades y daño causado.

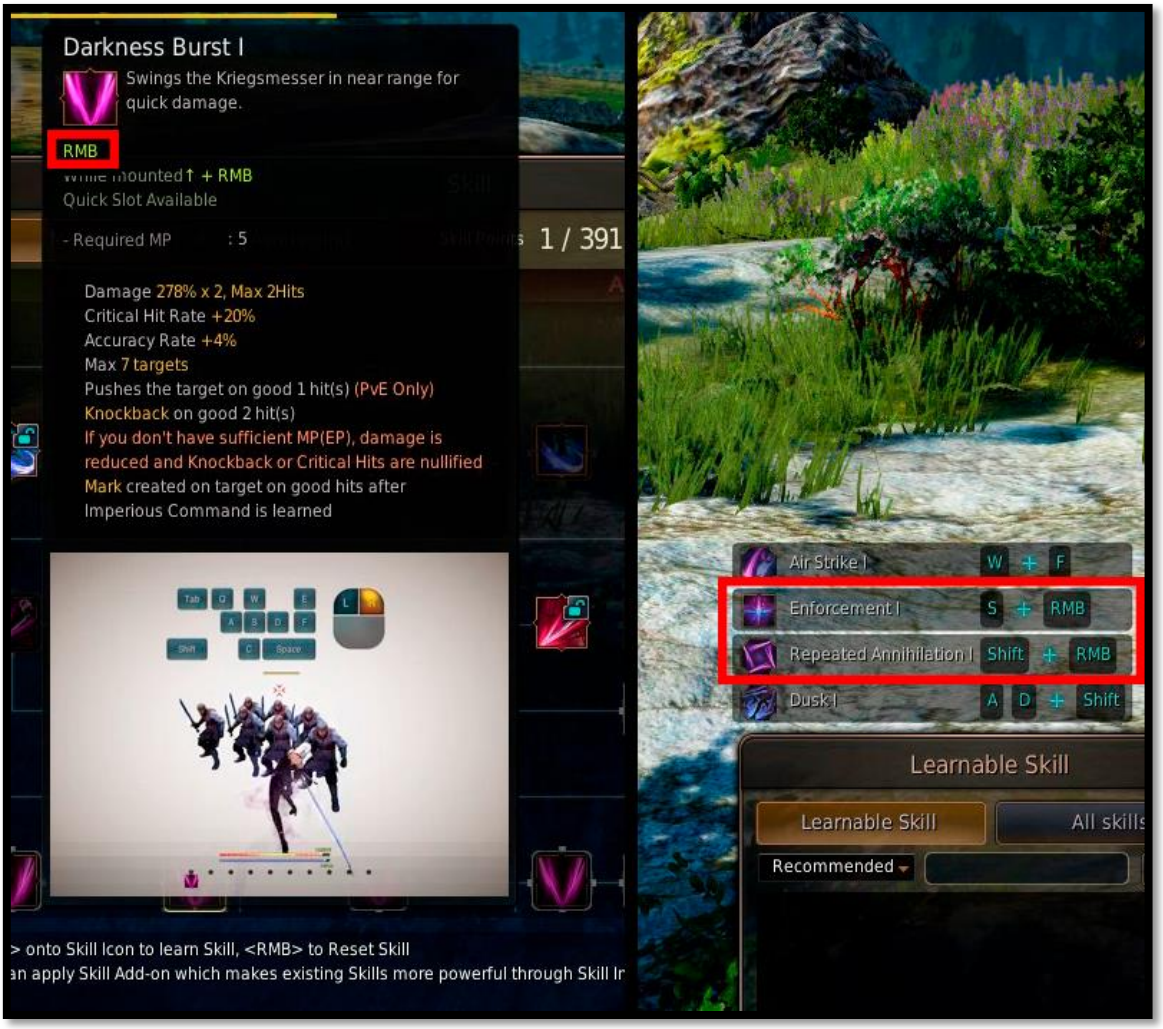


Figura 42 Black Desert Online, un MMORPG con un combate basado en combos

Fuente: elaboración propia

En la figura 42, a la izquierda vemos que la habilidad *Darkness Burst I* se usa apretando el botón derecho del ratón (*Right Mouse Button* o RMB). Pero, si miramos a la derecha, en el recuadro rojo vemos que si apretamos el botón derecho del ratón al mismo tiempo que la tecla S usaremos la habilidad *Enforcement I*. Si optamos por pulsar tecla *SHIFT* en nuestro teclado

cuando hacemos clic derecho con el ratón, usaremos, en su lugar, la habilidad *Repeated Anhilitation I*.

Al principio de esta sección se hablaba de como este sistema evoluciona el anterior. Esta evolución causa que se potencien tanto las ventajas como las desventajas. El jugador tiene más habilidades y combinaciones poderosas, pero esas habilidades hay que diseñarlas e implementarlas en el juego y a mayor número de elementos en un juego, más difícil es de equilibrar.

Además, el sistema cuenta con una desventaja más: la dificultad del juego aumenta de manera cuadrática. En la figura 41 vemos como *Wizard of Legend* cuenta con seis huecos para habilidades en la parte inferior izquierda, aunque solo cuatro están en uso en la captura. En este juego, esas seis habilidades serían simplemente seis habilidades. Ocurre al contrario con *Black Desert Online*, donde esas seis habilidades resultarían en seis habilidades básicas y treinta y seis combinaciones, dando un total de cuarenta y dos habilidades diferentes que el jugador tiene que memorizar tanto sus efectos como la secuencia para usarlas.

Sistema de combate centrado en la habilidad del jugador

Aunque este sistema es esencialmente un sistema basado en habilidades, la diferencia entre ellos radica en que las "habilidades" de este sistema son simplemente acciones que el jugador puede realizar como atacar con una espada, esquivar con una voltereta y bloquear con un escudo.

Si bien pasar de un hechizo que lanza un meteorito cubierto en llamas que explota, haciendo temblar la pantalla y eliminando a multitud de enemigos a una simple estocada con espada puede parecer un retroceso, el atractivo de este sistema es la humanización del personaje y la simpleza del sistema. Este sistema es, además, capaz de transmitir múltiples sensaciones que al jugador, siendo las siguientes algunas de ellas:

- Inmersión del usuario en el juego: en el mundo real, nadie puede lanzar hechizos como el mencionado anteriormente, pero cualquiera puede atacar con una espada. El jugador se sentirá más identificado con el personaje que puede hacer lo mismo que él.

- Sensación de desafío: el jugador puede ser obligado a luchar contra un enemigo enorme cuando solo disponga de una espada y un escudo. El jugador se sentirá desafiado ante el hecho de tener que enfrentarse a un poderoso enemigo con apenas dos objetos.

Figura 43 Pelea contra el Demonio del Asilo, el primer jefe de Dark Souls. El jugador solo dispone de una espada rota.

Fuente: www.youtube.com por el usuario ScarletThread [36]

- Sensación de triunfo: al superar un desafío, el jugador se sentirá poderoso y habilidoso por poder pasar un reto que al principio parecía imposible.

Pero estas sensaciones no son el único punto fuerte de este sistema de combate. Es un hecho que un juego tiene que estar equilibrado sin importar los sistemas que lo formen. Estos sistemas pueden influir en la dificultad de equilibrado en mayor o menor medida.

En un combate influyen los objetos, los enemigos y las habilidades del personaje, cada uno equilibrado por separado. Al contrario que un sistema basado en habilidades, ahora hablamos de acciones, que no pueden ser fundamentalmente equilibradas ya que son la parte más pequeña del controlador del personaje, lo que hace que este sistema sea el que menos influencia tiene en el equilibrio del juego general.

Finalmente, lo divertido que sea un juego o los retos que presenta vienen dados por el diseño de niveles y los enemigos presentes y por el desafío que el juego en sí presenta al jugador.

Conclusiones

El sistema de combate basado en la habilidad del jugador es el que más se acerca a las especificaciones del juego y el que ofrece más sensaciones que yo, como desarrollador, puedo explotar.

Objetivos

El objetivo principal de este trabajo es desarrollar un videojuego del género ARPG, que tenga un acabado profesional lo más profesional posible y que sirva como carta de presentación en mi porfolio.

Para llevarlo a cabo se recorrerán todas las etapas del desarrollo de un videojuego necesarias, tal y como lo hacen los estudios profesionales del mercado. Aunque no parto de cero con algunos de los programas como Unity o Adobe Photoshop, espero poder reforzar mis conocimientos y habilidades en aquellos que conozco y aprender y controlar *software* que nunca he usado como Bosca Ceoil. Así, al finalizar el proyecto, tendré un amplio abanico de habilidades que me ayudarán en mi futuro profesional.

Objetivos específicos

- Realizar un estudio de los videojuegos actuales y pasados con características similares al que se desarrollará.
- Conocer y seguir las fases del desarrollo de un videojuego tal y como se realiza en las empresas profesionales.
- Diseño de personajes y mapas en dos dimensiones con un estilo PixelArt.
    - *Concept Art.*
    - Bocetado.
    - Diseño de una paleta de colores.
    - Dibujado.
    - Animación en el caso de los personajes.
    - Estudio de los colores.
- Composición de música y efectos.

- Desarrollo e implementación de la inteligencia artificial de enemigos y personajes no jugables.
- Equilibrado del juego y la economía de este.
- Desarrollo e implementación de un sistema de generación procedimental de mapas.

Metodología

Para el desarrollo de un software determinado, se pueden seguir distintas metodologías. El tipo de proyecto o el tamaño del equipo de desarrolladores influyen en la determinación de la metodología más adecuada. Las principales son [37, 38, 39, 40, 41]:

- **Metodología en cascada**: considerada como la forma tradicional de desarrollar software. Consiste en un modelo lineal rígido compuesto por fases secuenciales: requerimientos, diseño, implementación, verificación y mantenimiento en este orden. Cada fase tiene que estar finalizada antes de avanzar a la siguiente. El principal inconveniente es que el método no contempla una forma de volver hacia atrás en el modelo para cambiar alguna especificación o alguna directiva de diseño. Equipos pequeños y software con un propósito único claro se benefician de esta metodología.
- **Metodología de Desarrollo Rápido de la Aplicación (*Rapid Application Development* o RAD)**: esta metodología condensa el proceso de desarrollo para producir sistemas de alta calidad con poca inversión. Está formado por cuatro fases: planteamiento de los requerimientos, diseño del usuario, construcción y corte, en ese orden, aunque las fases de diseño del usuario y construcción se repiten tantas veces como sea necesario, hasta que el cliente aprueba que todos los requerimientos se han cumplido [42]. Esta metodología es buena para aquellos equipos de un tamaño moderado y que tienen un usuario objetivo definido.
- **Metodología de desarrollo ágil**: dentro de esta metodología se inscriben varios métodos (*Scrum, Crystal, Extreme Programming, Feature-Driven Development*, etc.) que, aunque ligeramente diferente entre ellos, todos cumplen con los fundamentos de la metodología ágil: minimizar riesgos como bugs o excesos en el presupuesto haciendo uso del desarrollo iterativo, cada una añadiendo un pequeño incremento de las funcionalidades del producto. Además, cada fase o iteración del desarrollo se revisa constantemente, lo que permite localizar bugs y errores y solucionarlos en el momento o en la siguiente iteración. Esta metodología beneficia a aquellos equipos pequeños, en

los que el coste de comunicación no es muy grande y con un producto de escala incremental.

Ya que el proyecto consiste en el desarrollo de un videojuego con diferentes mecánicas y funcionalidades, he estimado que la adopción de una metodología ágil es la más adecuada para este proyecto.

Además, dentro de los diferentes métodos que forman la familia de metodologías ágiles, me he decantado por una metodología de *Feature-Driven Devolpment* o Desarrollo Enfocado a Funcionalidades. Esta forma de aplicar la metodología ágil se diferencia principalmente en la duración de las iteraciones, que es más reducido (normalmente de dos semanas, aunque pueden extenderse hasta a cuatro semanas), para acomodarse a los pequeños incrementos en funcionalidades que cada una añade al producto. Algunas de las ventajas del Desarrollo Enfocado a Funcionalidades son [43, 44]:

- El equipo de desarrolladores no malgasta el tiempo desarrollando soluciones innecesarias que luego serán desechadas.
- Cada componente del producto final ha sido aprobado y satisface los requerimientos.
- Rápida detección de errores y respuesta a estos.
- Entrega y monitorización continua.
- Alta importancia de la simplicidad y eliminación del trabajo innecesario, para acomodarse a las iteraciones.

Sin embargo, esta metodología también tiene desventajas como la intensidad del trabajo, al tener que seguir el ritmo de las iteraciones de duración reducida y el coste de comunicación entre miembros del equipo, pero, al contar con un solo desarrollador, esta desventaja, que es la principal de esta metodología, no supone un problema para el proyecto.

Organización de las iteraciones

Para organizar el trabajo dentro de las iteraciones se ha utilizado una metodología Kanban. En una metodología Kanban, todo el proceso es visible desde la definición de una tarea hasta su finalización. Una tarea esta siempre visible bajo la sección correspondiente a su actual estado de desarrollo. La ventaja de esta metodología es que limita la cantidad de trabajo para no sobrecargar a los participantes.

Las fases de desarrollo son, fundamentalmente, las siguientes:

1. **Pendiente**: una tarea esta recién creada o el trabajador al que se le ha encargado aún no ha empezado a trabajar en ella.
2. **En progreso**: una tarea está siendo desarrollada por el trabajador o los trabajadores encargados.
3. **Finalizada**: la tarea ha sido terminada. Aquellos que trabajaron en ella cogerán ahora una nueva tarea en la que trabajar.

A esta lista se pueden añadir fases adicionales dependiendo del tipo de trabajo a realizar. En el desarrollo de software, se suelen añadir las siguientes fases:

- **En prueba**: una tarea está acabada, pero aún no se ha dado por terminada. Esto se hará después de probar la funcionalidad y confirmar que su funcionamiento es correcto.
- **Revisión**: una tarea que había sido dada por terminada ha dado problemas o se puede mejorar.

HacknPlan

Para el desarrollo de este TFG se ha usado la herramienta online HacknPlan [45], la cual permite aplicar una metodología Kanban de manera gratuita y con una sencilla e intuitiva interfaz. El punto fuerte de esta herramienta es que está orientada al desarrollo de videojuegos y proporciona la capacidad de clasificar tareas atendiendo al aspecto del juego con el que se relaciona, como puede ser diseño, programación o arte.

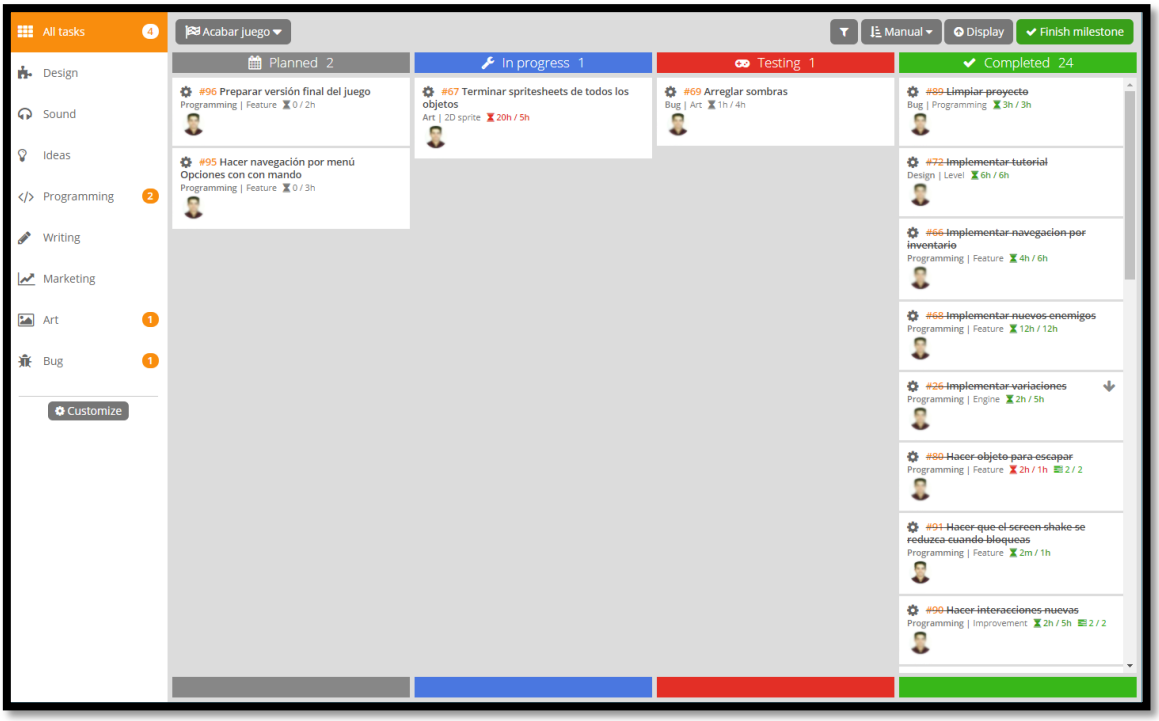


Figura 44 Tareas listadas y organizadas atendiendo a su estado

Fuente: elaboración propia

HacknPlan permite, también, dividir las tareas en *milestones* o hitos, lo que permite seguir la organización en iteraciones de la metodología ágil principal.

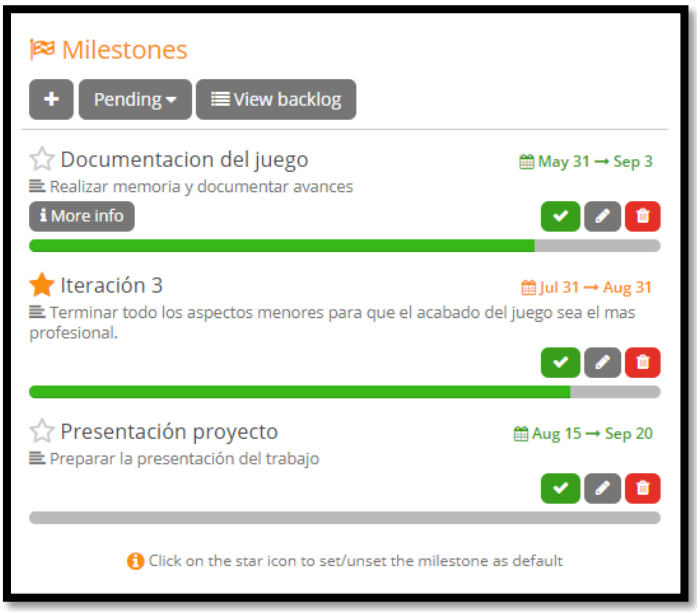


Figura 45 Algunos hitos del proyecto.

Fuente: elaboración propia

A la hora de definir una tarea se han seguido unos pasos concretos:

1. Darle un nombre identificativo: este nombre ha de explicar de manera clara y sencilla el objetivo de la tarea. Para este fin, todos los nombres se componen de un verbo en infinitivo seguido del sustantivo correspondiente, por ejemplo: *Programar enemigos*.
2. Definir un hito en el que incluir la tarea.
3. Asignar un responsable de la tarea.
4. Categorizar la tarea: las tareas se pueden agrupar en varias categorías como programación o arte. Además, es importante diferenciar entre las subcategorías posibles. De este modo, podemos separar una tarea de *Programación/Arreglar bug* de una de *Programación/Característica*, ya que son ámbitos diferentes
5. Estimar el tiempo de desarrollo: basándonos en anteriores tareas parecidas y en la experiencia en el ámbito de la tarea, intentamos estimar el tiempo que llevara terminar la tarea.
6. Añadir una descripción: si la tarea es extensa, con el fin de evitar un título excesivamente largo, añadiremos una descripción sobre aspectos de la tarea que haya que añadir o bien anotaciones para recordarlas en el momento de su implementación.
7. Añadir subtareas: si la tarea es extensa y se puede desglosar en tareas podemos indicarlas y más adelante ir tachando subtareas conforma se vayan realizando.

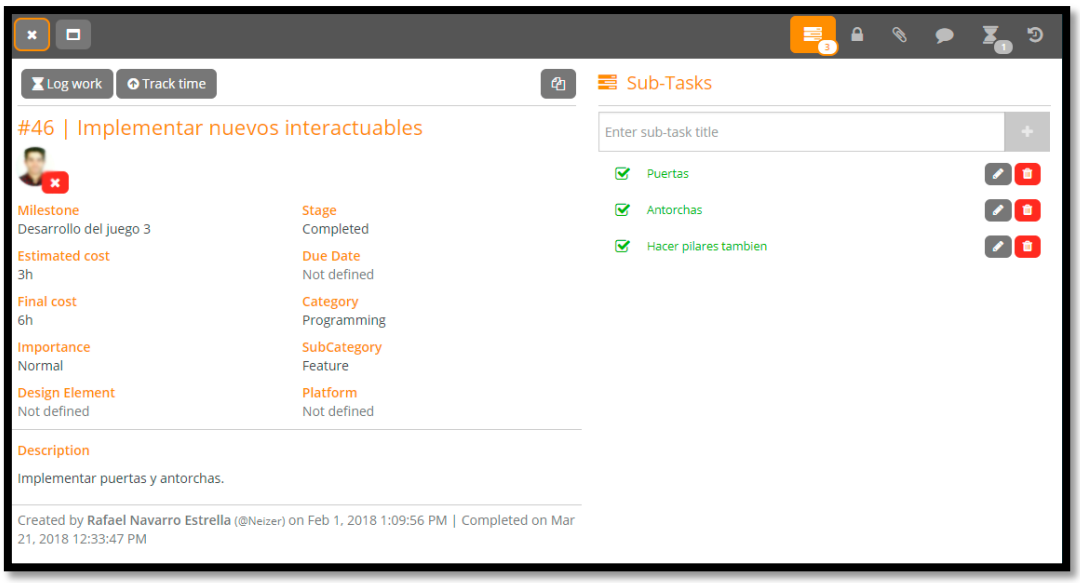


Figura 46 ejemplo de una tarea terminada

Fuente: elaboración propia

Herramientas utilizadas para el desarrollo

Unity

Figura 47 Imagotipo de Unity

Fuente: www.evensi.com

Como ya se concluyó en la sección *Motores de desarrollo de videojuegos*, se ha utilizado Unity como motor para el desarrollo para facilitar y acelerar el trabajo a la hora de integrar el apartado grafico con el código y la lógica del juego.

Visual Studio

Figura 48 Imagotipo de Visual Studio

Fuente: www.theregister.co.uk

Se ha utilizado Visual Studio como entorno de desarrollo integrado (IDE o Integrated Development Environment) para la creación y edición del código fuente.

Photoshop

Figura 49 Isologo de Adobe Photoshop

Fuente: es.wikipedia.org

Se ha usado Adobe Photoshop para la creación del apartado gráfico como personajes, objetos o entornos.

Audacity

Figura 50 Imagotipo de Audacity

Fuente: foro.vozidea.com

Audacity ha sido usado para editar y modificar algunos sonidos para que se aproximasen más a los requerimientos deseados del videojuego. Siempre se han editado sonidos cuya licencia lo permitiese.

Bosca Ceoil

Figura 51 Logotipo de Bosca Ceoil

Fuente: terrycavanagh.itch.io

Bosca Ceoil es un programa poco conocido que permite crear música y efectos de manera sencilla. Se ha utilizado para crear algunos efectos y canciones cortas.

Adobe Premiere

Figura 52 Isologo de Adobe Premiere

Fuente: www.pcmag.com

Para editar videos como el tráiler del juego se ha utilizado Adobe Premiere, ya que es una herramienta muy potente de material audiovisual.

Cuerpo del trabajo

A la hora de desarrollar software, el primer paso a seguir es definir unas especificaciones que dicho software debe cumplir. En el caso de un videojuego, estas especificaciones se recopilan en un GDD o *Game Design Document*, el cual, como su nombre indica, es un documento que registra todas las decisiones de diseño de un juego, como mecánicas, controles o aspecto gráfico.

GDD

Nombre del juego

El juego que se desarrollará se llamará *Into the Crypt.*

Información general del juego

En esta sección se expondrán algunos detalles generales de *Into the Crypt*.

Concepto del juego

Into the Crypt es un juego de rol en el que el jugador tendrá que recorrer los niveles de una mazmorra generada de manera procedimental, por lo que cada partida es diferente. Los niveles son intrincados, llenos tanto de habitaciones amplias y repletas de enemigos como pasillos estrechos y tortuosos. Las mazmorras, además de una gran variedad de enemigos, tienen también numerosas trampas.

A lo largo de los niveles, el jugador podrá encontrar objetos que le ayuden a fortalecerse, o bien tirados en el suelo o bien dentro de cofres. El jugador también podrá fortalecerse ganando experiencia al vencer enemigos, lo cual, finalmente, le llevará a subir de nivel.

Aquellos que jueguen a *Into the Crypt* se encontrarán inmersos en una oscura mazmorra rebosante de peligros, que harán que sienta la soledad y el riesgo que sufre el personaje principal.

Género del juego

Into the Crypt será un juego del género ARPG (Action Role Playing Game).

Público objetivo

Aquellos jugadores que disfruten los juegos RPG, que les guste ser desafiados y en general la exploración de intrincadas mazmorras procedurales encontrarán en *Into the Crypt* un juego ideal. Por eso, este juego va dirigido principalmente a todos ellos, aunque cualquiera podrá jugar y pasar un buen rato, independientemente de sus gustos.

Resumen del ciclo del juego

El ciclo o flujo del juego se compone de las fases que el juego tiene y como estas se desarrollan en un orden específico. Al llegar a la última, la primera debería suceder a esta. El ciclo de juego en *Into the Crypt* es muy sencillo:

1. El jugador llega a un nuevo nivel.
2. El jugador decide explorar un camino de la mazmorra intentando llegar a la salida.
3. El jugador lucha contra enemigos, recoge objetos e interacciona con el entorno.
4. El jugador llega al final del camino, que puede ser:
    a. Un final muerto. El jugador tendrá que volver sobre sus pasos para explorar otro camino.
    b. El jugador ha llegado a la salida del nivel, encontrará un pequeño botín y unas escaleras que le llevarán al inicio del siguiente nivel, que será más difícil que el actual.

Si en cualquier momento el jugador pierde la partida, se le presentará las hazañas que ha conseguido a lo largo de su aventura. Estas son:

- Piso máximo alcanzado.
- Enemigos eliminados.
- Monedas de oro conseguidas.
- Valor monetario de los objetos del inventario.
- Oro total conseguido.

Look and feel

El *look and feel* hace referencia a como tiene que ser el aspecto visual del juego y que sensaciones tiene que transmitir. En *Into the Crypt*, el mundo tiene que ser siniestro y tener un aspecto lúgubre, ya que el jugador estará dentro de una cripta repleto de enemigos no-muertos.

Además, las habitaciones estarán oscuras excepto por la luz del jugador y alguna antorcha que pueda estar presente.

Into the Crypt tiene que hacer sentir al jugador que está solo, en una cripta oscura y peligrosa, es decir, tiene que transmitir sensación de terror y riesgo.

Jugabilidad y mecánicas

En esta sección se detallan características fundamentales del núcleo del juego.

Jugabilidad

Dentro de la jugabilidad podemos diferenciar dos aspectos diferentes: jugabilidad como solo usabilidad y jugabilidad relativa a la calidad de la experiencia de juego [46].

1. Jugabilidad como solo usabilidad: este aspecto se define mediante una combinación de interfaz y controles que determinan como de usable es un juego. En *Into the Crypt*, la interfaz se ha diseñado para que sea clara y lo menos intrusiva posible, mostrando en pantalla solo la información más importante. Información menos relevante estará presente en submenús y pantallas alternativas. En cuanto a los controles y gracias a que *Into the Crypt* tiene un número limitado de acciones que el jugador puede realizar, hace que estos sean simples e intuitivos. Tanto la *interfaz* como los *controles* se detallan más adelante.
2. Jugabilidad relativa a la calidad de experiencia de juego: este aspecto se refiere a la calidad de la sensación o experiencia que se transmite al controlar el juego. En *Into the Crypt,* el jugador puede esperar un control del personaje fluido, con un combate justo en el que los instantes de ataque, bloqueo y evasión están refinados manualmente. Además, el jugador tiene múltiples objetos con los que interactuar y amplias mazmorras que explorar.

Mecánicas

Las mecánicas en un juego son todas aquellas formas de interacción entre el jugador y el mundo del juego. Estas interacciones van restringidas por una serie de reglas. Las mecánicas definen como se juega a un juego. Las mecánicas de *Into the Crypt* son:

- **Atacar**: el jugador podrá atacar con su arma principal. Además, los ataques se pueden encadenar formando combos. El máximo de ataques de un combo viene dado por el tipo de arma del jugador.
- **Bloquear con escudo**: el jugador podrá usar su escudo para bloquear una cantidad de daño de un ataque. La cantidad de daño bloqueada depende tanto de la resistencia del escudo como del tiempo de reacción del jugador: bloquear en el momento justo será más efectivo que simplemente cubrirse todo el tiempo con el escudo.
- **Beber pociones**: el jugador podrá tomar una poción para recuperar una cantidad de salud.
- **Dar esprints***:* el jugador podrá dar un *sprint*, ganando una gran cantidad de velocidad durante un corto periodo de tiempo, lo que le permitirá esquivar enemigos y trampas.
- **Ganar de experiencia y subir de nivel**: el jugador gana experiencia derrotando enemigos. Al alcanzar el máximo de experiencia subirá de nivel, recargando su vida y energía. Cada nivel es más difícil de alcanzar que el anterior.
- **Interactuar con el entorno**: en el mundo de *Into the Crypt* hay múltiples objetos que el jugador podrá usar. Estos son:
    - **Objetos de equipamiento**: equipamiento como armas, armaduras o escudos que haya en el suelo pueden ser recogidos por el jugador, almacenándolos en el inventario.
    - **Abrir puertas**: algunas salas estarán detrás de puertas cerradas. El jugador podrá abrir estas puertas para acceder al interior.
    - **Abrir cofres**: a lo largo de la mazmorra habrá cofres en algunas habitaciones. Estos cofres contendrán objetos de equipamiento, monedas de oro y en ocasiones, pociones. Los cofres podrán ser de tres niveles, conteniendo en su interior mejor botín conforme mejor sea el cofre.
    - **Encender antorcha**s: el jugador podrá encender antorchar que encuentre en la mazmorra y que esté apagada. Las antorchas encendidas proporcionarán luz.
    - **Cambiar de piso**: una vez llegado al final de un nivel, el jugador encontrará una escalera con la que tendrá que interactuar para poder avanzar al siguiente nivel.
- **Recoger objetos del suelo**: a diferencia que los objetos de equipamiento, objetos varios como pociones o monedas de oro no necesitan que el jugador pulse un botón para interactuar con ellos, será suficiente con tocarlos.

- **Gestionar el inventario y equipamiento**: el jugador podrá abrir su inventario en cualquier momento y navegar por él, inspeccionando los objetos que haya conseguido. El jugador podrá equipar o desequipar un objeto a voluntad.
- **Observar el mapa**: el jugador podrá mirar en el mapa para ver las habitaciones que ya ha investigado, podrá, de esta forma, buscar caminos que haya pasado por alto o rutas inexploradas.
- **Pausar el juego**: el jugador podrá detener el ciclo del juego, lo que le llevará a un menú desde el que podrá retomar la partida, volver al menú o salir del juego.
- **Navegar por los menús**: si el juego no se encuentra dentro de su ciclo principal, significa que el jugador está en los menús. El jugador podrá navegar por ellos libremente.

Opciones del juego

En *Into the Crypt*, el jugador dispone de un amplio abanico de posibilidades para que pueda personalizar la experiencia del juego.

- **Armas**: existen tres tipos de armas, cada una con un combo de ataques diferentes y estilos de combate diferentes.

Tipo de arma	**Posibles armas**	**Combo máximo**	**Beneficios**
Cortantes	Espadas, dagas, sables …	3	Armas equilibradas y versátiles.
Contundentes	Hachas, mazas, martillos …	2	Armas lentas, pero más poderosas.
Punzantes	Lanzas, partisanas, jabalinas …	1	Armas seguras y con mayor alcance.

Tabla 6 Tipos de armas en Into the Crypt

- **Habitaciones con tesoro**: en algunas habitaciones habrá habitáculos con enemigos y tesoros. El jugador puede decidir si entrar y arriesgarse para conseguir el botín o evitar entrar.
- **Diversas aproximaciones al combate**: el jugador puede decidir si combatir de una forma más agresiva recibiendo algo de daño a cambio o por el contrario, centrarse en bloquear y esquivar ataques enemigos, mientras espera a la oportunidad perfecta para atacar, minimizando así riesgos a costa de tiempo.

Limitaciones en el juego

Las mecánicas de un juego se rigen por normas que limitan el comportamiento del jugador o el desarrollo del propio juego. Algunas de las limitaciones presentes en *Into the Crypt* son:

- El jugador cuenta con una cantidad de vida y energía. Si la vida llega a cero, la partida habrá terminado. Dar esprints consume energía. El jugador solo dispone de energía suficiente para dar dos esprints, evitando así que supere los niveles en menos tiempo del deseado. Al pasar un corto intervalo de tiempo tras gastar energía, esta empezará a recargase poco a poco.
- El jugador nunca recibirá objetos de nivel superior al suyo. Los objetos de equipamiento se clasifican en niveles: un nivel de objeto equivale a dos niveles del jugador. Para evitar que el jugador consiga objetos demasiado fuertes cuando sea de nivel bajo, estos se restringirán para potenciar la sensación de progreso.
- El jugador podrá llevar un máximo de tres pociones y tendrá un inventario con veinte casillas para llevar el equipamiento que encuentre en su aventura.
- El jugador dispondrá de cuatro posibilidades para equiparse objetos, siendo estos la cabeza, el cuerpo, el arma principal y el escudo.

Historia, trasfondo y personajes

A continuación, se hará una introducción al mundo de *Into the Crypt* en el que el jugador se verá inmerso.

Historia y narrativa

La historia de *Into the Crypt* es la siguiente:

"Adéntrate en un mundo de fantasía, cuyos habitantes intentan vivir de manera apacible a pesar de los monstruos que aterrorizan todas las regiones. Para lidiar con los monstruos surgió una nueva profesión: el aventurero. Los aventureros son personas valientes y siempre dispuestos a ayudar. Sin embargo, en las últimas semanas, los monstruos han aumentado su número y han empezado a atacar de manera más organizada y temible. Alguien tiene que estar dirigiéndolos.

No hay suficientes aventureros como para lidiar con la amenaza. Solo el héroe legendario de la profecía, el único que puede salvar al mundo, será capaz de derrotar al líder de los monstruos.

Pero ese no eres tú. Solo eres un simple aventurero que ha aceptado una misión para purgar una cripta cercana a cambio de un puñado de monedas de oro. Buena suerte.”

Mundo del juego

Into the Crypt se desarrollará dentro de una cripta abandonada. Esta tendrá las siguientes características relativas al aspecto:

- Paredes de ladrillos grises, algo desgastados.
- Suelo de fríos adoquines o de madera mohosa y vieja.
- Pilares de diferentes tamaños
- Nichos dentro de las habitaciones de la cripta, de los que surjan enemigos.
- Gran cantidad de trampas y enemigos.

Por último, la cripta estará oscura, permitiendo al jugador solamente un pequeño radio de visión.

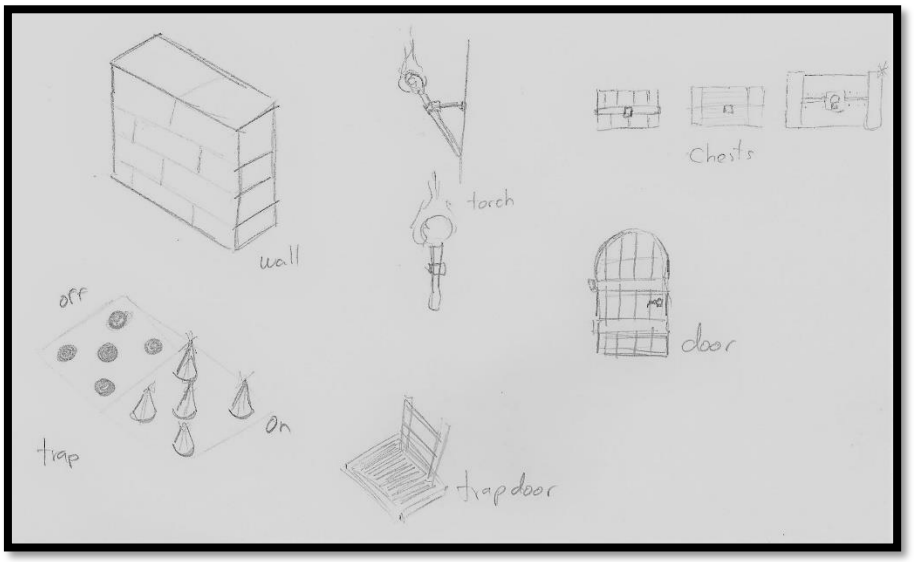


Figura 53 Boceto de los elementos

Fuente: elaboración propia

Figura 54 Boceto de ejemplo de una habitación de la cripta

Fuente: elaboración propia

Personajes

En *Into the Crypt* haremos una diferenciación entre el personaje protagonista y los enemigos.

El personaje protagonista es aquel que el jugador controlará. En este juego, el protagonista no es un héroe, ni es elegido por nadie ni va a cumplir ninguna profecía; es simplemente un aventurero cualquiera que ha decidido aceptar una misión para conseguir algo de dinero con lo que sobrevivir hasta el día de mañana. Para enfatizar esta faceta, el personaje será un varón de estatura media, con pelo castaño y de complexión estándar. El personaje empezará con una sencilla túnica de color rojo, una vieja espada oxidada y un endeble escudo de cuero. Recordemos que este aventurero cualquiera, después de volver de una aventura, apenas habrá conseguido dinero para pagarse una cena caliente y una noche en una posada.

Figura 55 Boceto del personaje

Fuente: elaboración propia

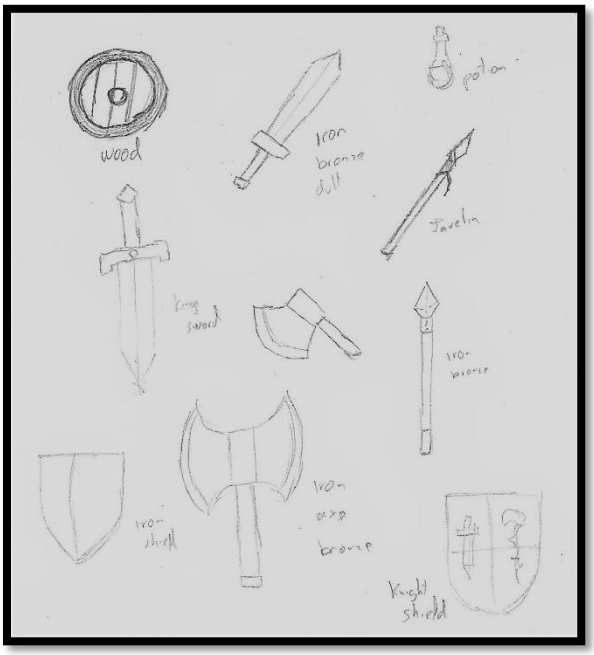


Figura 56 Bocetos de algunos objetos que el protagonista podrá usar

Fuente: elaboración propia

Para los enemigos, habrá una predominancia de seres no-muertos, ya que la misión del protagonista es ir a una cripta a eliminarlos. Entre los enemigos encontraremos varios tipos:

- ***Zombies***: enemigos básicos. Perseguirán al jugador para intentar golpearle o morderle. Habrá varios tipos de *zombies*, que añadirán dificultad y variedad a los niveles:
    - Cadáver fresco: Un *zombie* más rápido debido a su reciente fallecimiento y resurrección. Debido a esto, su piel tiene un color más humano y será más rápido y fiero. Irá vestido con poca ropa: unos pantalones y un arnés.
    - *Zombie* común: Debido a la cantidad de tiempo que ha pasado entre que pereció y fue resucitado este *zombie* tiene una piel de color verde oliva y va ataviado con unas ropas ajadas. Causa daños moderados al atacar y se mueve a una velocidad media.
    - Cadáver hinchado: En vida, este *zombie* murió ahogado. Debido a esto, su piel es de un tono verde oscuro y su cuerpo estará hinchado, confiriéndole mayor tamaño y resistencia, pero haciendo que se mueva a menor velocidad. La ropa que porta estará deshecha por el agua en la que pareció, pero ira cubierto de algas, corales y animales marinos como percebes.

Tipo de zombie	**Resistencia**	**Ataque**	**Velocidad**
Cadáver fresco	Baja	Alto	Alta
Zombie común	Media	Medio	Media
Cadáver hinchado	Alta	Medio	Baja

Tabla 7 Resumen de las estadísticas que diferencian a los zombies

Figura 57 boceto de los zombies

Fuente: elaboración propia

- **Espíritus**: seres similares a elementales de fuego. Tienen aspecto de llama. Al igual que con los *zombies*, habrá diferentes tipos y cada tipo actuará de manera diferente.
    - Elemental de fuego menor o chispa: Una pequeña llama de color anaranjado que deja un rastro de llamas allá por donde pasa.
    - Elemental corrupto: Un elemental de fuego que ha sido corrompido por la magia de ultratumba. Esto le confiere un color verde lima y el poder de lanzar proyectiles ígneos.
    - Fuego fatuo: Un elemental de fuego que ha consumido un alma en pena. De color azul, lanza proyectiles de fuego en todas las direcciones.

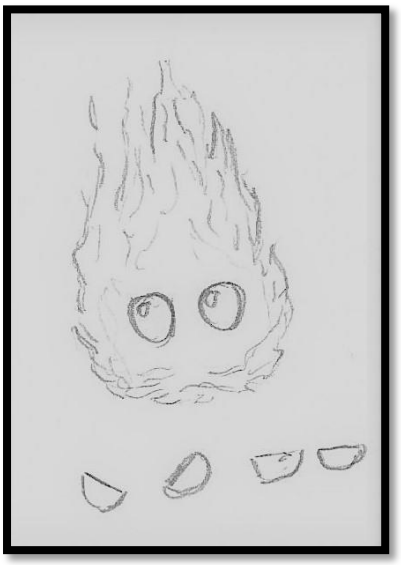


Figura 58 boceto de los espiritus

Fuente: elaboración propia

- **Babosas**: enemigos peligrosos que consumen los objetos que se encuentran a su paso y se dividen al morir.

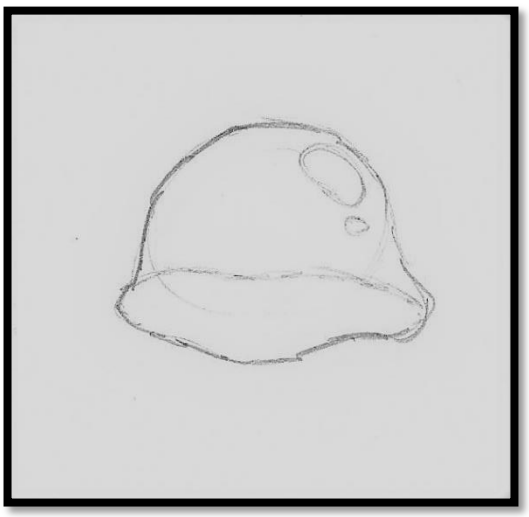


Figura 59 boceto de una babosa

Fuente: elaboración propia

- **Almas errantes**: enemigos fantasmagóricos, vestidos con una túnica ajada y portadores de una linterna que emite luz del otro mundo. Atraviesan paredes y potencian a sus enemigos a la vez que intentarán atacar al jugador.

Figura 60 Boceto de un alma errante

Fuente: elaboración propia

- **Esqueletos de aventureros**: enemigos fuertes y rápidos, ya que son los restos de aventureros que perecieron en la cripta. Se moverán a alta velocidad hacia el jugador para acabar con él.

- **Esqueletos arqueros**: Esqueletos de otros aventureros aun armados con sus armas y armaduras. Este en particular porta un arco, lo que le permite disparar flechas al jugador.
- **Esqueletos espadachines**: El aventurero al que este esqueleto pertenecía era diestro en el combate cuerpo a cuerpo. Porta consigo, además de su armadura, un escudo con el bloquear y una temible arma cuerpo a cuerpo.

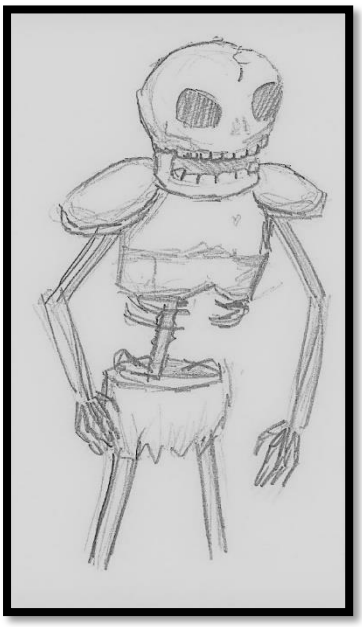


Figura 61 Bocetos de los esqueletos

Fuente: elaboración propia

- **Nigromante**: jefe final que aparecerá cada cierto número de niveles. Lanzará proyectiles mágicos al jugador e invocará enemigos de los listados anteriormente para derrotar al protagonista.

Figura 62 Boceto del nigromante

Fuente: elaboración propia

Para la versión final se intentará tener el máximo de estos personajes incluidos en el juego.

Diseño de niveles

En *Into the Crypt*, los niveles estarán generados procedimentalmente. Sin embargo, estos niveles se componen de habitaciones que sí serán diseñadas a mano. Las habitaciones deberán cumplir una serie de especificaciones:

- Las habitaciones tienen que crear oportunidades para combatir con los enemigos, ya sea en zonas despejadas y amplias o con lugares estrechos y con obstáculos.
- Las habitaciones tienen que permitir un flujo cómodo para que recorrerlas no sea tedioso. Además, el jugador tiene que poder elegir entre varios caminos para recorrer una habitación y en ocasiones, debe poder elegir entre un camino seguro y uno más arriesgado pero que le otorgue una recompensa.
- Las habitaciones tendrán que tener recovecos en los que esconder enemigos, objetos o cofres.
- En las habitaciones tendrá que haber espacios seguros a los que el jugador pueda retirarse para tomar una poción a salvo o recargar energía.

Para evitar la repetitividad en las habitaciones habrá que diseñar suficientes para que no se repitan demasiado.

Niveles

A continuación, se mostrarán algunos diseños para las habitaciones, siguiendo la siguiente leyenda:

Marca	**Objeto representado**
Cuadrado de color negro	Paredes
Cuadrado de color blanco	Suelo
Cuadrado de color verde	Conectores de habitaciones
Cuadrado de color rosa	Jugador
Cuadrado de color rojo	Enemigos
Cuadrado de color naranja	Siguiente nivel
Cuadrado de color azul	Trampas
Cuadrado de color morado	Objetos (equipamiento, cofres, etc.)
Cuadrado de color turquesa	Puertas

Tabla 8 Leyenda para los diseños de habitaciones

Tanto la habitación inicial, en la que el jugador aparece al empezar un nuevo nivel, como la final, desde la cual avanza al siguiente, serán siempre iguales.

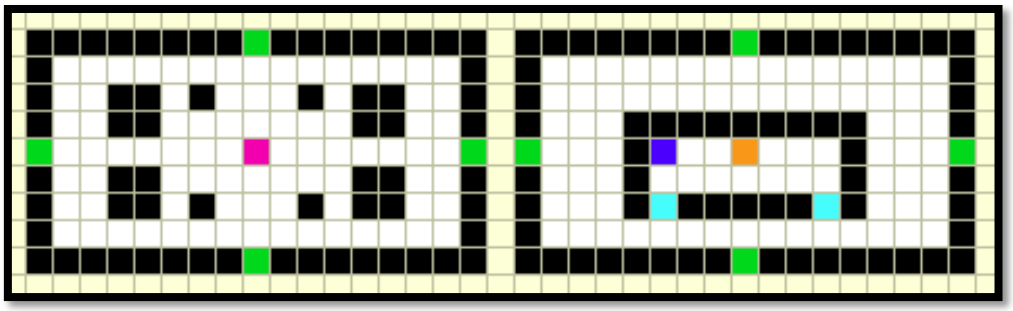


Figura 63 La habitación inicial, a la izquierda y la final a la derecha

Fuente: elaboración propia

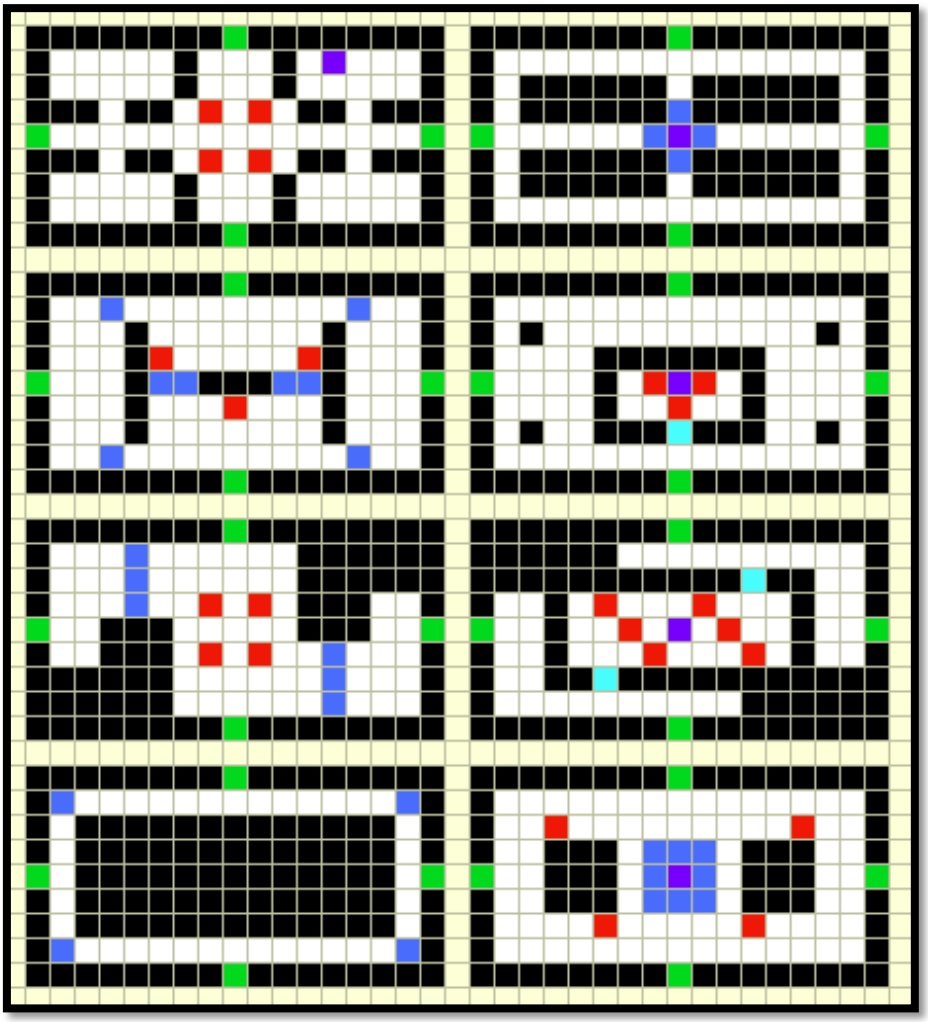


Figura 64 Ejemplos de habitaciones

Fuente: elaboración propia

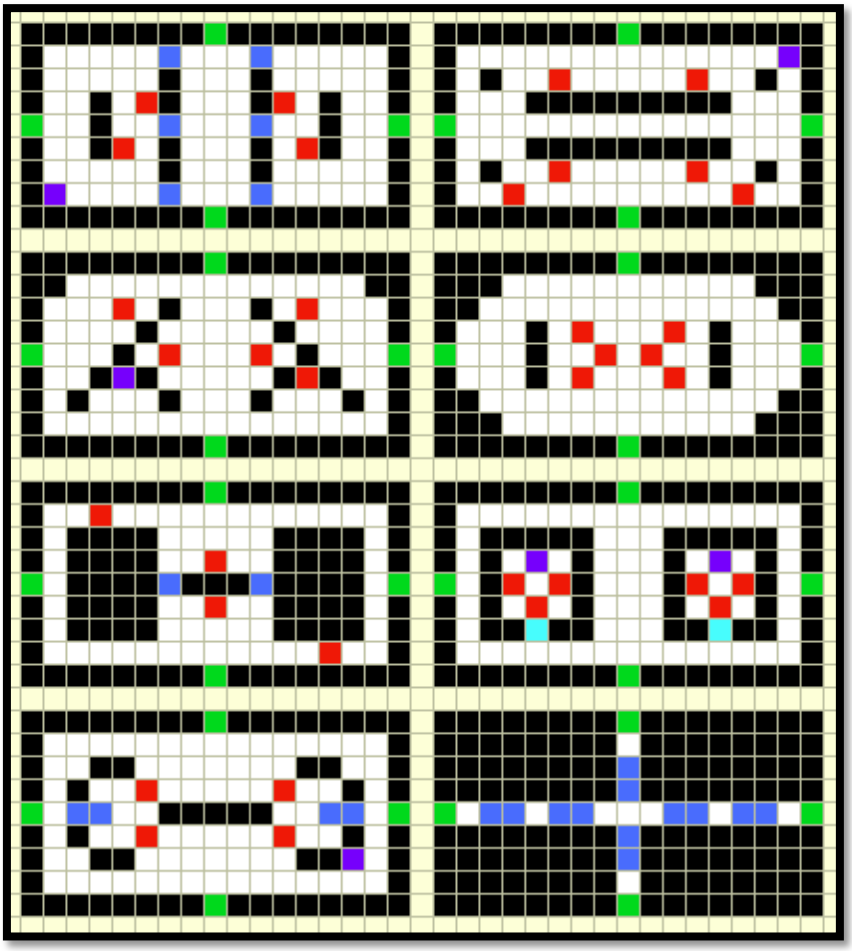


Figura 65 Ejemplos adicionales de habitaciones

Fuente: elaboración propia

Tutoriales

Into the Crypt contará con un único tutorial que enseñará al jugador los fundamentos del juego y qué acciones podrá realizar con el personaje. El mapa consistirá en una serie de pasillos y habitaciones de tamaño reducido. En cada una de estas habitaciones habrá una mecánica nueva por aprender. Aparecerán, además, textos no demasiado intrusivos que indiquen al jugador qué acción realizar para superar la habitación y mediante qué botones llevarla a cabo.

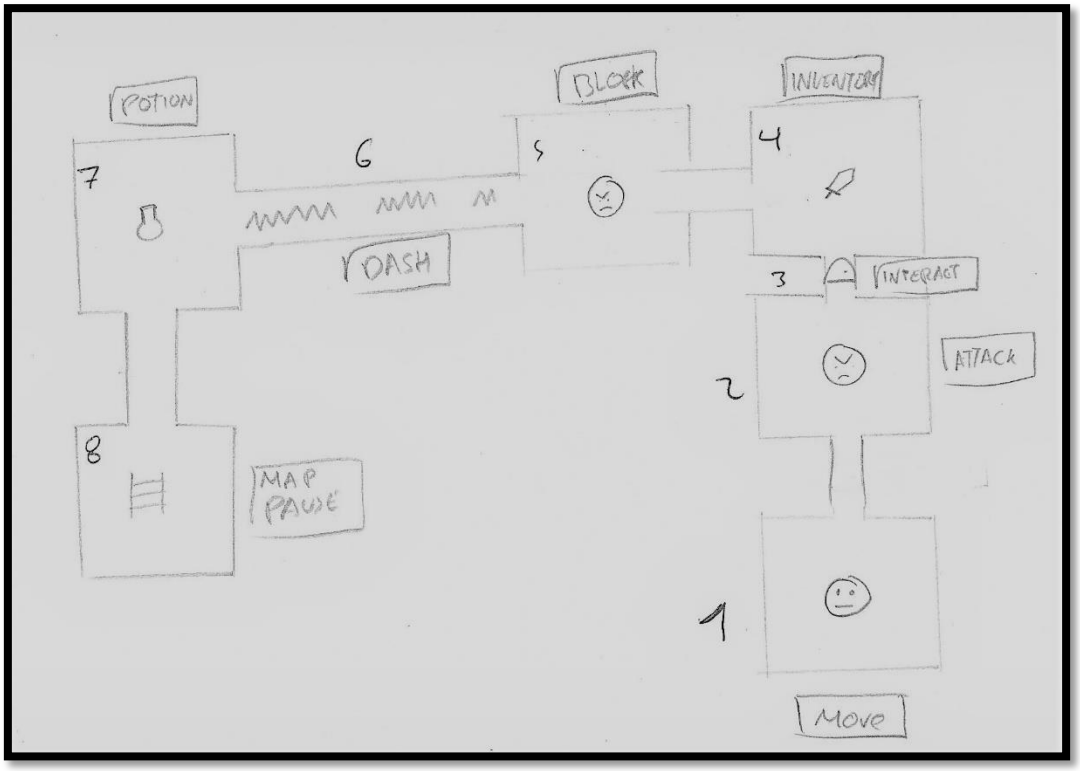


Figura 66 Boceto del mapa de tutorial

Fuente: elaboración propia

El flujo del mapa del tutorial será el siguiente:

1. El jugador empezará en una habitación vacía, con un pasillo al norte. Aparecerá un texto que le indicará como desplazarse.
2. El jugador recorrerá un pasillo y llegara a una sala con un enemigo que le perseguirá, pero nunca atacara. Un texto le indicará al jugador como atacar para derrotar a este enemigo. La salida estará al norte.
3. Tras derrotar a su primer adversario, el jugador se encontrará en un pasillo en el que una puerta le cierra el paso. Un texto le indicará como interactuar con ella para avanzar.
4. El jugador llegará, entonces, a una habitación en la que una espada le esperará en el suelo. El jugador podrá recogerla de la misma forma que abrió la puerta. Un texto le indicará como abrir el inventario para poder equiparla.
5. El jugador avanzará hasta una sala con un enemigo inmóvil, que no parará de atacar. Un texto le enseñara al jugador a bloquear, con el fin de que pruebe a parar los ataques de este enemigo. El jugador podrá derrotarlo para avanzar o simplemente pasar de largo.
6. Tras esto, el jugador llegara a un largo pasillo lleno de trampas. Podrá atravesar una trampa de manera segura cuando esta esté escondida, llegando a una zona segura antes

de otra zona con más trampas. En este momento, un texto le enseñará a esprintar, lo que le permitirá atravesar el grupo de trampas. Llegará, entonces, a una zona segura, ante la cual habrá una gran cantidad de trampas. El jugador, al intentar cruzarlas, recibirá daño y llegará sin toda la vida a la siguiente sala.

7. En esta sala, una poción esperará al jugador y un texto le indicará como beberla para recuperar la salud.
8. La siguiente habitación será la última. En esta, habrá una escalera con la que el jugador podrá acabar el tutorial y empezar a jugar en un nivel normal. Además, dos textos le enseñaran como abrir el mapa y como pausar el juego.

Prueba de conocimientos

La prueba de conocimientos se refiere a como el jugador pone en práctica los conocimientos adquiridos. Conforme el jugador avanza en el juego, el jugador usará lo aprendido en el tutorial para avanzar en su aventura. Sin embargo, el jugador tiene que seguir aprendiendo a medida que juega.

Dentro del nivel del tutorial:

- El jugador, tras aprender a interactuar con una puerta, aprenderá que puede interactuar de igual forma con una espada del suelo y con las escaleras del final del nivel.
- Tras aprender a pelear, el jugador podría derrotar al segundo enemigo en vez de intentar bloquear sus ataques.

En los niveles normales:

- El jugador descubrirá que algunas antorchas están encendidas y otras apagadas y que puede interactuar con las apagadas para encenderlas y que estas emitan luz.
- El jugador descubrirá que hay más enemigos aparte de los básicos encontrados en el tutorial y que estos poseen habilidades muy diferentes.
- El jugador descubrirá la existencia de cofres y como estos contienen en su interior objetos y dinero.

Interfaz

La interfaz de *Into the Crypt* tendrá que ser lo más sencilla posible para que esta no sea intrusiva y permita al jugador control del mundo (posición de los enemigos, trampas, vías de escape, etc.) en todo momento.

Sistema visual

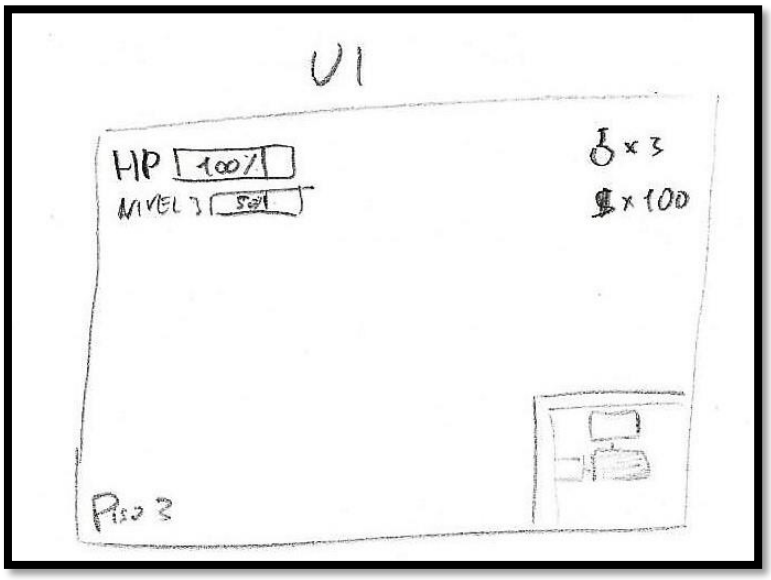


Figura 67 Boceto de la interfaz del juego

Fuente: elaboración propia

En el juego se presentará la información mediante una interfaz muy sencilla, situada en las esquinas para no interferir con el juego.

- Arriba a la izquierda tendremos dos barras que indicarán la cantidad de salud del jugador. Además de un número que representará el nivel actual del jugador. Aparecerán también unos indicadores en forma de rayo que indicarán de cuanta energía dispone el jugador para esprintar.
- Arriba a la derecha, el jugador podrá ver cuántas pociones y cuánto dinero tiene.
- Abajo a la derecha, el jugador podrá ver un mini mapa con su posición actual.
- Abajo a la izquierda aparecerá el piso en el que se encuentra el jugador en ese momento.

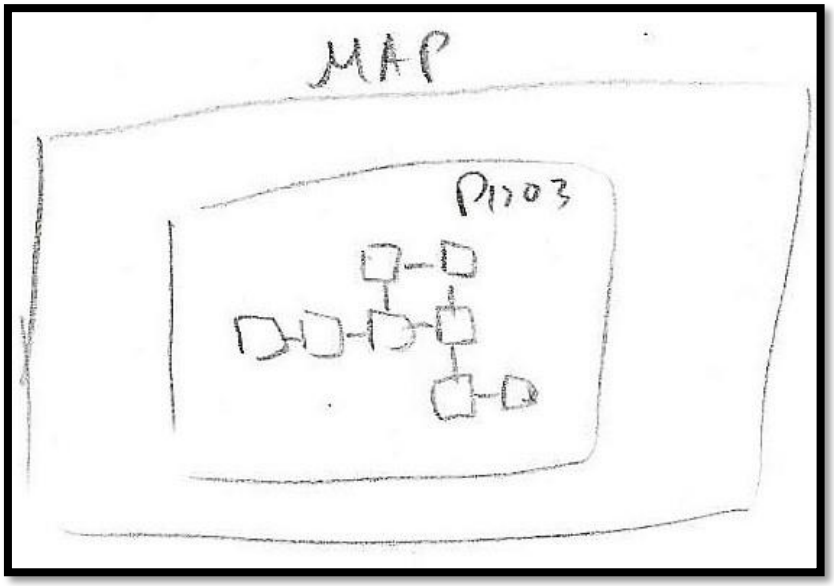


Figura 68 Boceto de la interfaz del mapa

Fuente: elaboración propia

Cuando el jugador abre el mapa, se mostrará en el centro de la pantalla y ocupando casi la totalidad de esta un cuadrado en el que aparecerá el mapa completo, ya que en el mini mapa no aparecerán todas las habitaciones, sino solo las más cercanas.

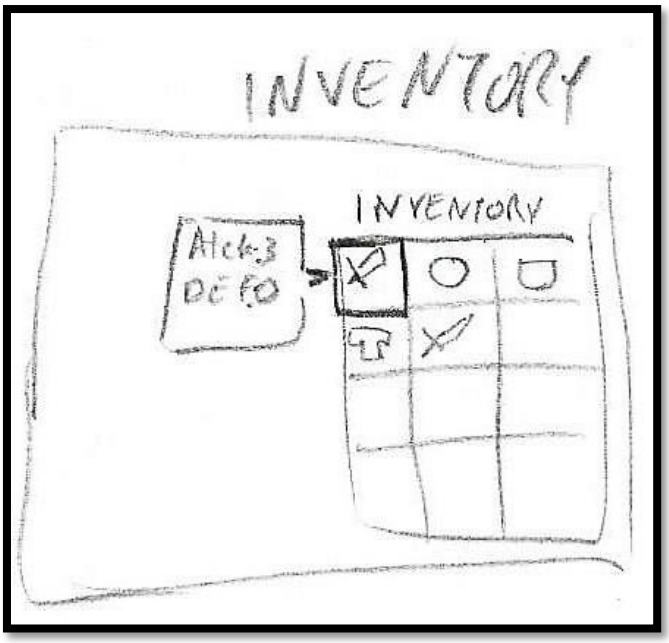


Figura 69 Boceto de la interfaz del inventario

Fuente: elaboración propia

Al abrir el inventario, aparecerá en la parte derecha de la pantalla una cuadricula con los objetos que el jugador posea. El jugador podrá navegar por esta cuadricula, inspeccionando los objetos.

Al pararse en una casilla en la que haya un objeto, aparecerá un mensaje indicando las estadísticas del objeto como el daño de ataque o la defensa que otorga.

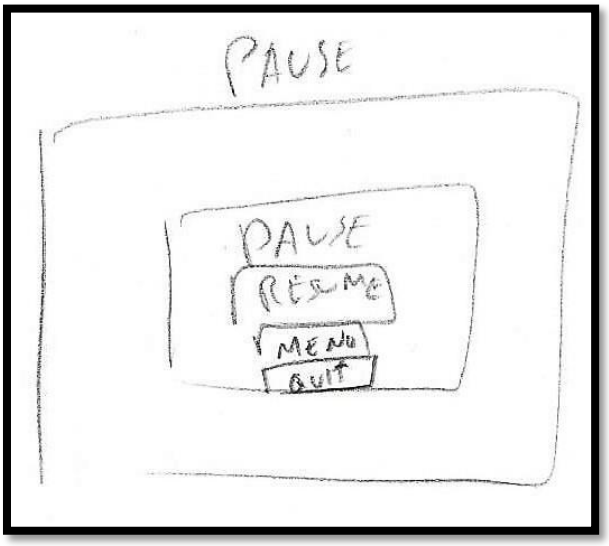


Figura 70 Boceto de la interfaz del menú de pausa

Fuente: elaboración propia

Al pausar el juego, un menú con tres opciones será presentado al jugador:

- Retomar partida: permitirá al jugador volver a jugar donde lo había dejado.
- Menú principal: permitirá al jugador volver al menú principal, descartando todo su progreso.
- Salir: cerrará el juego.

El botón de "Retomar partida" será más grande que los demás, para evitar errores al hacer clic en él y para animar al jugador a seguir jugando.

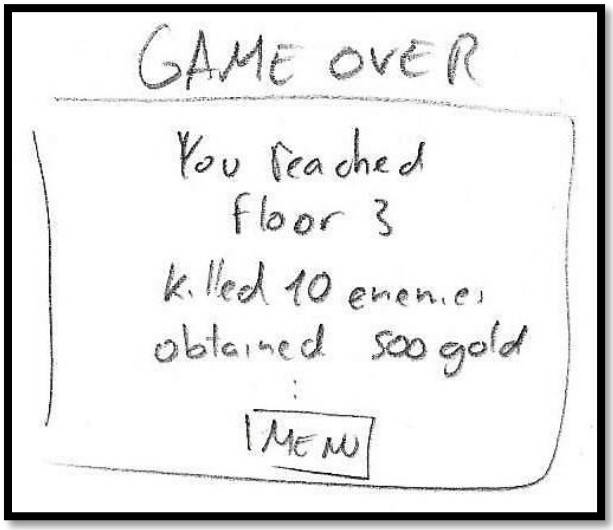


Figura 71 Boceto de la interfaz del final del juego

Fuente: elaboración propia

Al perecer en la mazmorra, se mostrará una pantalla con las estadísticas del jugador. Las estadísticas mostradas son aquellas que se especificaron en la sección *Resumen del ciclo del juego*. Se le dará al jugador la opción de volver al menú tras ver el resumen de su partida.

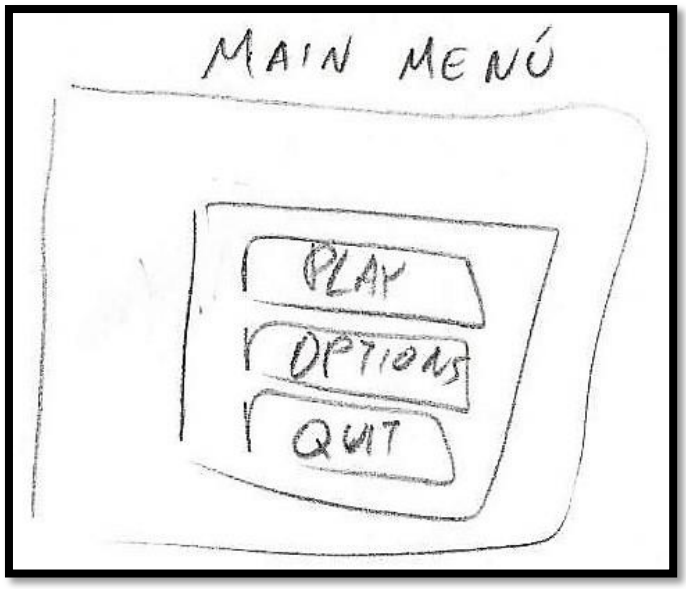


Figura 72 Boceto de la interfaz del menú principal

Fuente: elaboración propia

El menú principal será muy similar al menú de pausa. Variará en las opciones que se muestren al jugador y que el tamaño de los botones será igual para todos los botones.

Controles

El jugador podrá jugar a *Into the Crypt* tanto con mando como con teclado. Ninguno de los dos métodos tiene que hacer que le juego sea más difícil que el otro método. Además, el jugador podrá navegar por lo menús haciendo, también, uso del ratón. Los controles serán los siguientes:

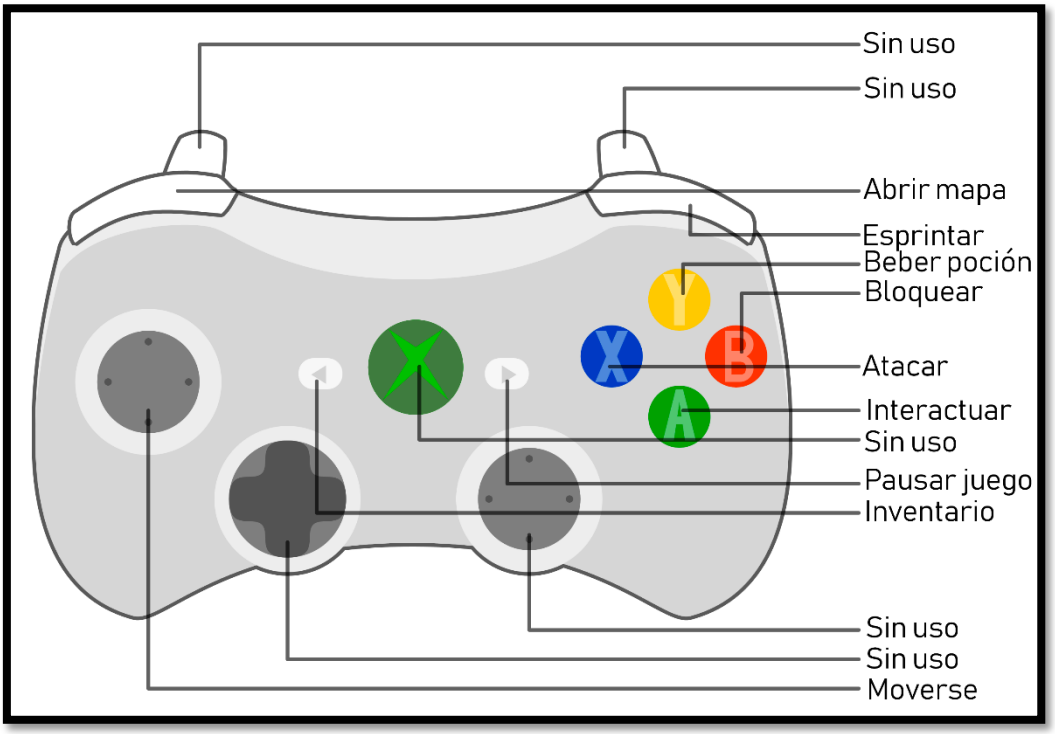


Figura 73 Controles de Into the Crypt para mando

Fuente: elaboración propia, a partir de la plantilla disponible en www.clker.com

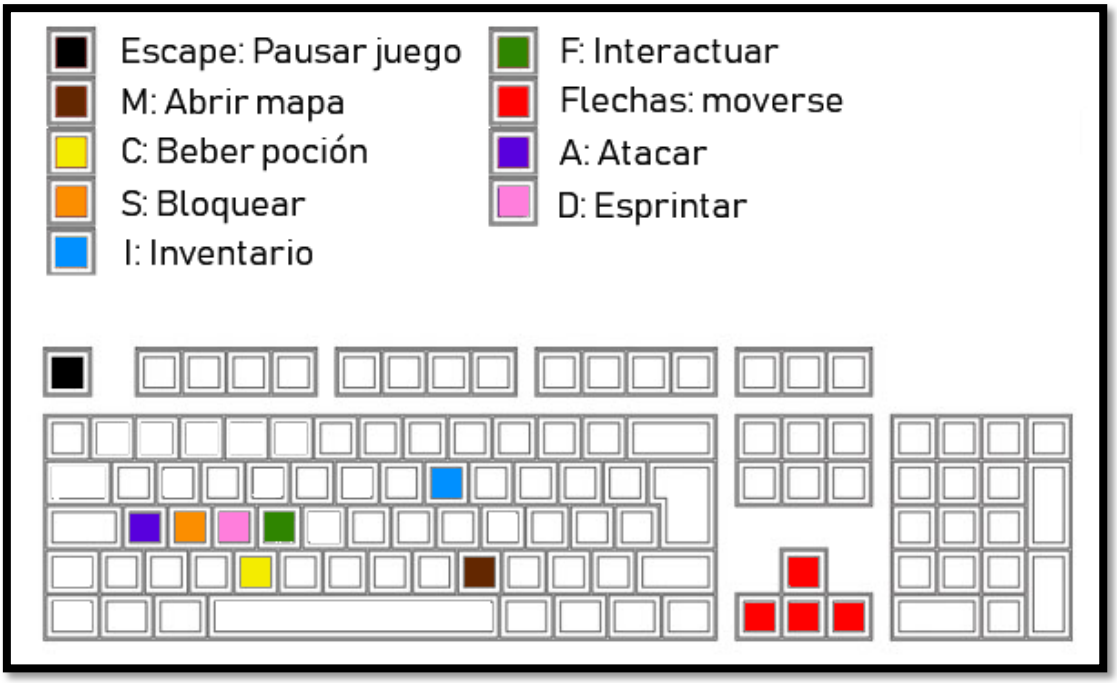


Figura 74 Controles de Into the Crypt para teclado

Fuente: elaboración propia a partir de la plantilla disponible en gaminggrad.com

Sonido, música y efectos

En cuanto al aspecto sonoro de *Into the Crypt*, haremos una distinción entre música y efectos de sonido:

- La música es aquella que se escucha de fondo mientras se juega. Ya que *Into the Crypt* está ambientado en una cripta oscura. La música será tétrica y terrorífica. Habrá varios temas que transmitirán menor o mayor sensación de peligro. Los que más transmitan esta sensación, sonaran al avanzar en el juego, cuando este aumente en dificultad.
- Para los efectos de sonido, habrá varios sonidos para cada efecto (ataque de espada, recoger dinero, etc.), de esta forma, el juego sonará más variado y no será tan repetitivo. Además, a estos sonidos se les pueden aplicar variaciones de tono de forma aleatoria para disminuir aún menos la sensación de repetición. Cada efecto tiene que tener un sonido o conjunto de sonidos fácilmente diferenciable de los demás, para evitar confusiones y aumentar la claridad del juego.

Configuración

El jugador podrá acceder desde el menú principal a un menú secundario donde modificar la configuración del juego. En este menú el jugador podrá:

- Cambiar la resolución de pantalla.
- Cambiar entre pantalla completa o modo ventana.
- Cambiar el volumen del sonido general del juego.
- Silenciar todos los sonidos.
- Elegir entre tres configuraciones de gráficos preestablecidos para que el jugador los adapte a la potencia de su sistema.

Guía técnica

A continuación, se expondrán algunos detalles técnicos sobre el desarrollo de *Into the Crypt*.

Hardware y software para el desarrollo

El hardware usado para el desarrollo ha sido:

- Un ordenador de sobremesa y un ordenador portátil, ambos con diferentes especificaciones y con acceso a internet.
- Conjunto completo de periféricos: ratón, teclado, altavoces.
- Mando de XBOX 360 para probar el juego.

En cuanto al software se ha usado:

- Windows 10 como sistema operativo.
- Unity 2017.3.1 como motor de desarrollo.
- Visual Studio como editor de código.
- Adobe Photoshop para la realización del apartado gráfico.
- Bosca Ceoil y Audacity para el apartado sonoro.

Arquitectura del juego

El juego se desarrollará usando una arquitectura de componentes. En cuanto a la implementación, se definirán tres grandes apartados: los componentes, la máquina de estados del jugador y el árbol de comportamiento o *behaviour tree* de los enemigos.

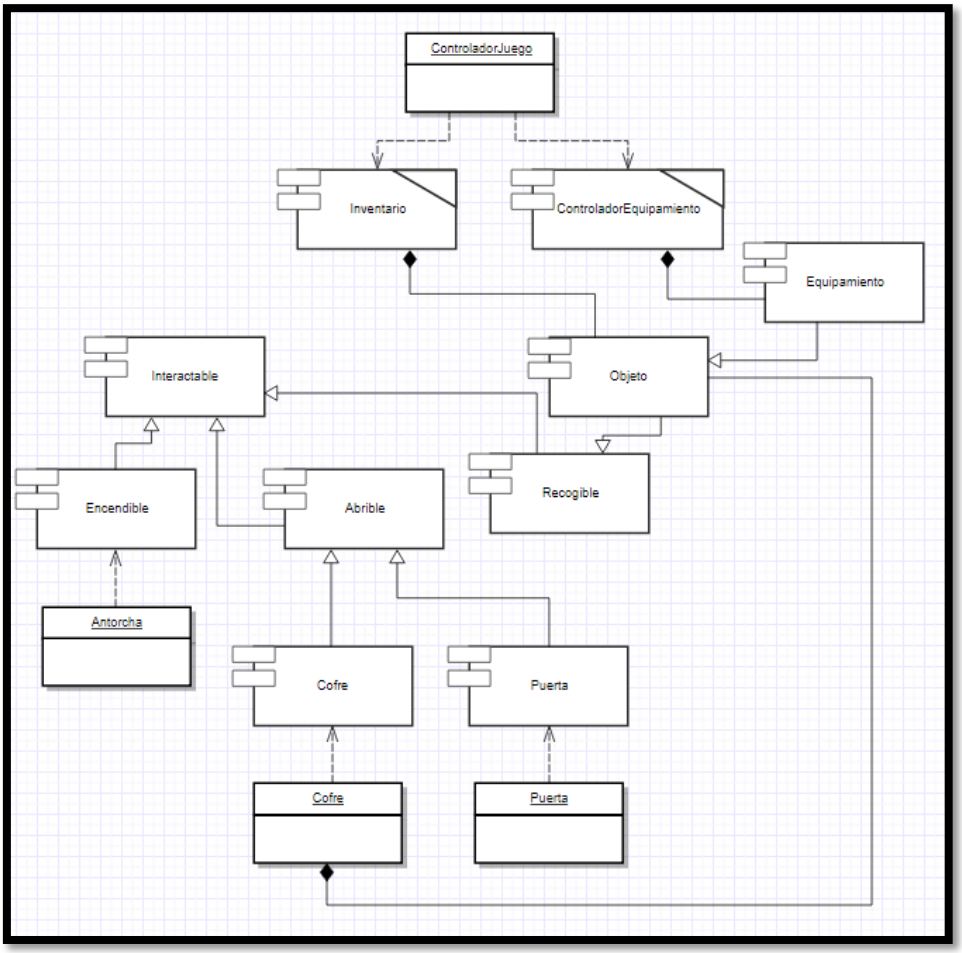


Figura 75 Primera parte del diagrama de objetos y componentes

Fuente: elaboración propia

En esta primera parte del diagrama tenemos los objetos con los que el jugador podrá interactuar. Todos ellos tendrán un componente que los defina (*Encendible*, *Abrible*, etc.) que heredará de *Interactuable*. Los objetos y el equipamiento irán, al interactuar con ellos, al inventario. El inventario se compondrá de una lista de objetos. Los objetos de equipamiento compondrán también la lista de equipo del jugador.

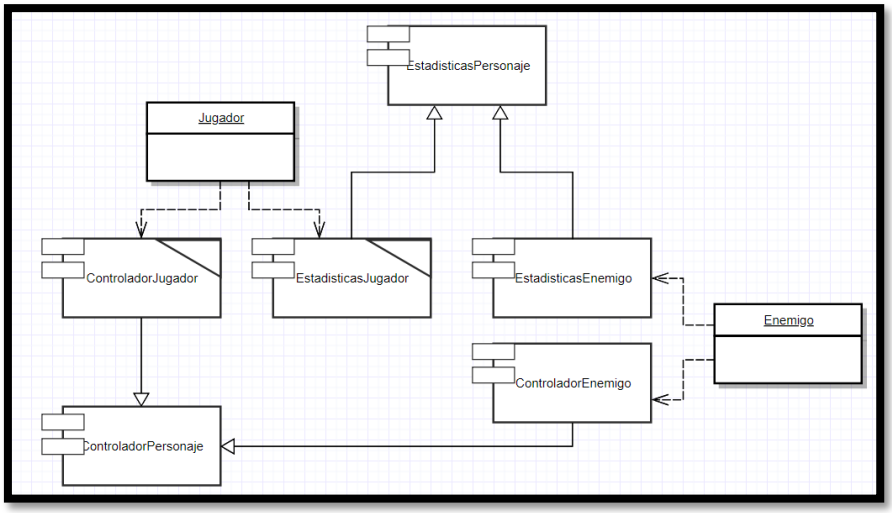


Figura 76 Segunda parte del diagrama de objetos y componentes

Fuente: elaboración propia

En este diagrama se presenta la implementación de los personajes. Todos los personajes, tanto el jugador como los enemigos, tendrán un componente que maneje las estadísticas y otro que se encargue de controlar sus acciones. Tanto los enemigos como el jugador comparten funciones, por lo que sus componentes heredaran de uno más genérico, aunque ciertas cosas puntuales se implementen de manera distinta más tarde. Lo mismo ocurre con las estadísticas.

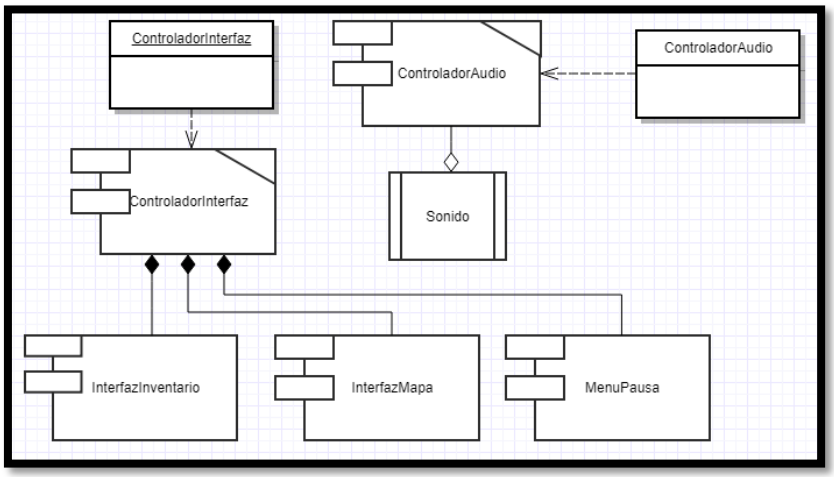


Figura 77 Tercera parte del diagrama de objetos y componentes

Fuente: elaboración propia

En este diagrama tenemos las funcionalidades del sistema. Un objeto *ControladorInterfaz* y otro *ControladorAudio* que se encargan de gestionar las diferentes interfaces del juego y el sonido del juego, respectivamente. El componente *ContorladorInterfaz* se compone del resto de interfaces.

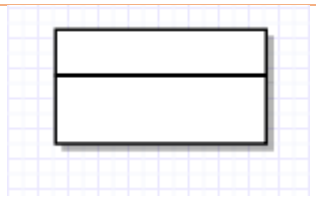
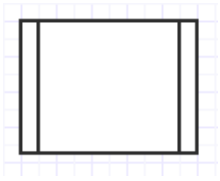
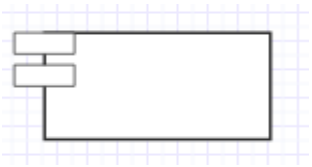
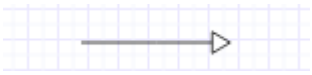
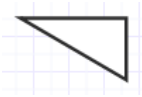
Símbolo	**Objeto representado**
	Objeto del juego (GameObject)
	Clase
	Componente
	Herencia
	Composición
	Agregación
	Dependencia
	Componentes que aplican el patrón *singleton*

Tabla 9 Leyenda del diagrama de objetos y componentes

Al seguir una arquitectura de componentes, hablamos de un diagrama de objetos y componentes en lugar de un diagrama de clases clásico. Hay que tener en cuenta que algunos componentes básicos como detectores de colisiones o transformaciones se han obviado del diagrama.

Vemos también que algunos de los componentes, como *ControladorJugador* aplica el patrón *singleton*. Esto hace que solo pueda haber una única instancia de este componente en un momento dado del juego. Además, cualquier componente que aplique este patrón podrá ser accedido desde cualquier parte del juego por cualquier componente, aunque no a cualquier coste, ya que estas referencias se almacenan en el *heap* en vez de en el *stack*, por lo que es más costoso acceder a ellas.

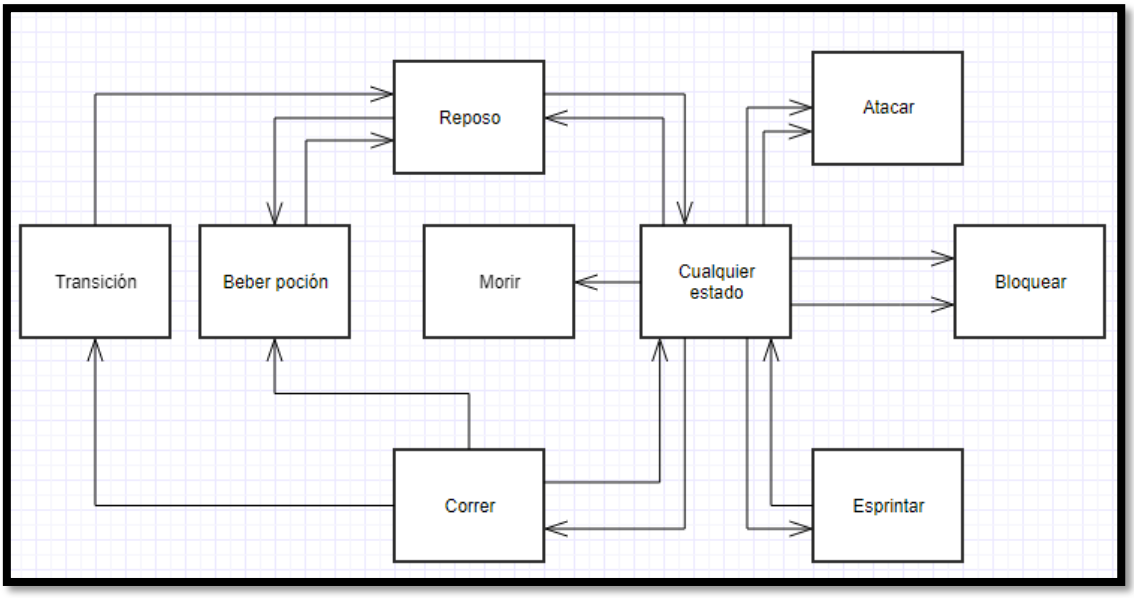


Figura 78 Diagrama de la maquina de estados del jugador

Fuente: elaboración propia

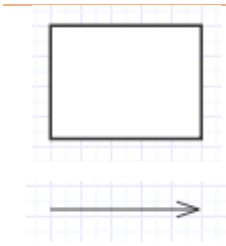
Símbolo	**Objeto representado**
	Estado Transición

Tabla 10 Leyenda del diagrama de la máquina de estados

En esta ocasión nos encontramos ante el diagrama de una máquina de estados para el control del jugador. En el centro encontramos un estado especial que solo existe en el diagrama: *Cualquier estado*. Este estado representa a todos aquellos que tienen transición desde y hacia él. Podemos ver, también, como el estado *Beber poción* solo puede ser accedido desde *Correr* y *Esperar* y al terminar siempre volverá a *Esperar*.

Los estados son fragmentos de código que definen un comportamiento de un sistema, en este caso del personaje protagonista. El funcionamiento de una máquina de estados es sencillo: una maquina solo puede tener un estado activo en un momento determinado. De esta forma, cuando el jugador esté atacando, solo ejecutará el código de atacar, por ejemplo.

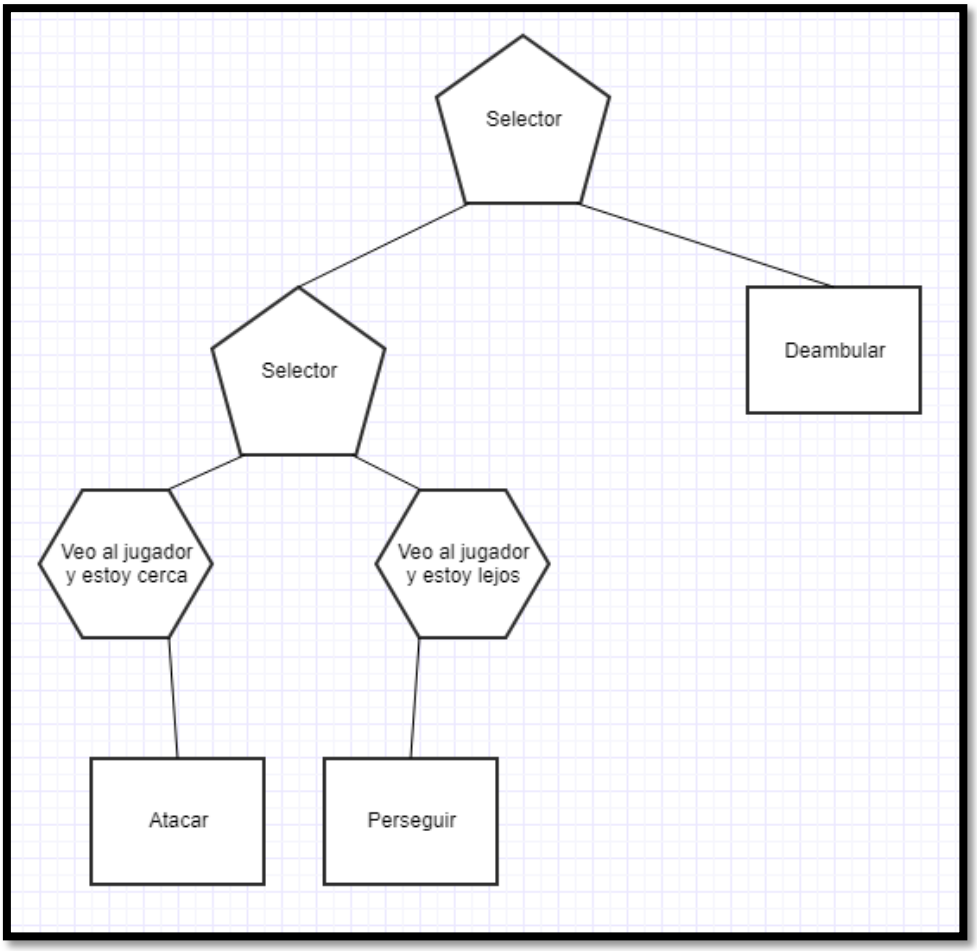


Figura 79 Arbol de comportamiento para la inteligancia artificial de los enemigos

Fuente: elaboración propia

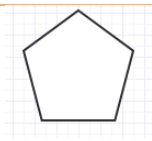
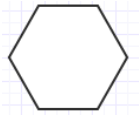
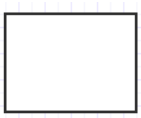
Símbolo	**Objeto representado**
	Nodo selector
	Nodo condición
	Nodo acción

Tabla 11 Leyenda para el árbol de comportamiento de los enemigos

El funcionamiento de un árbol de comportamiento se basa en un conjunto de nodos que forman el árbol y que se recorren siguiendo unas reglas que sus nodos definen.

- Un nodo selector buscará el primer nodo hijo que tenga éxito y lo ejecutará, tras lo cual parará la ejecución durante ese ciclo.
- Un nodo condición solo permite el avance si la condición que define se cumple.
- Un nodo acción es el propio comportamiento de una entidad de inteligencia artificial y contiene todas las acciones que definen el comportamiento.
- Existe otro tipo de nodo, secuencia, que ejecutará todos sus hijos hasta que uno falle o todos tengan éxito. Este nodo no se ha utilizado en el diseño de este árbol.

Con este diagrama podemos definir un comportamiento muy sencillo, pero a la vez realista y efectivo para los enemigos. Estos, como no-muertos que son, deambularan buscando una presa. Si en algún momento divisan al jugador, intentarán perseguirlo con el fin de acercarse lo suficiente como para poder atacar.

Inicio del desarrollo

Antes de empezar el desarrollo y una vez se tienen definidos los requisitos del juego en el GDD, se llevaron a cabo dos tareas para facilitar la primera etapa de desarrollo.

Diseño preliminar del juego

Cuando se prototipa un juego, se suelen utilizar formas geométricas de colores o *sprites* de otros juegos mientras se desarrolla la versión inicial del juego, pero, en este caso, se han usado algunos *sprites* de un prototipo anterior, sobre los que, además, se ha iterado más tarde. Cabe destacar que los *sprites* y en especial las animaciones que llevan desde el principio, como la de reposo o correr, distan mucho en calidad de las realizadas en las versiones más avanzadas del juego.

Figura 80 Sprites iniciales del jugador

Fuente: elaboración propia

Figura 81 Sprites iniciales de un zombie

Fuente: elaboración propia

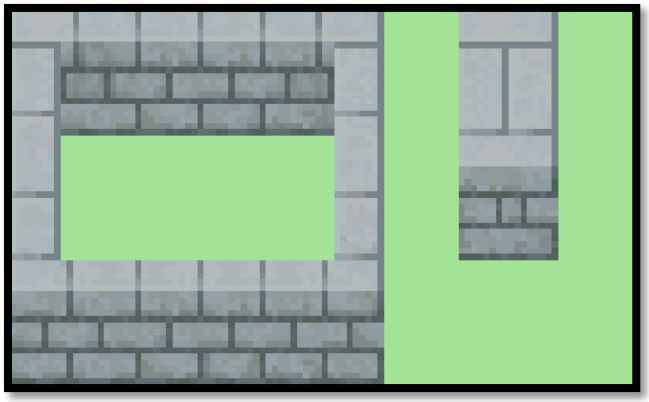


Figura 82 Tileset inicial para la mazmorra

Fuente: elaboración propia

Gracias a este inicio con ventaja en cuanto al aspecto visual, resultó más fácil mantener la moral alta durante el desarrollo y seguir implementando mecánicas.

Planificación

Al seguir una metodología de desarrollo iterativo enfocado a funcionalidades, se ha planificado el proyecto en iteraciones y se ha asignado a cada iteración un objetivo, que normalmente coincide con la implementación de una mecánica principal del juego.

Iteración	**Objetivo**
Iteración 1	Implementar estadísticas y controladores genéricos. Implementar máquina de estados del jugador y controlador de este.
Iteración 2	Desarrollar inteligencia artificial de los enemigos. Implementar un controlador para enemigos e integrarlo con la IA. Diseñar sistema de interacciones.
Iteración 3	Implementar objetos interactuables y sistema de objetos y equipamiento. Implementar menús e interfaz.
Iteración 4	Implementar sistema de generación de mazmorras.
Iteración 5	Terminar contenido del juego. Desarrollar sistema de sonido. Desarrollar un nivel de tutorial.
Iteración 6	Corrección de errores y preparación para la publicación del juego.

Tabla 12 Planificación del proyecto en iteraciones

A continuación, se mostrará la evolución del juego a lo largo de las iteraciones y, al final de cada sección sobre una iteración, se listarán los problemas que dicha iteración ha supuesto y que soluciones se han aplicado.

Primera iteración

El objetivo principal de esta iteración es implementar un sistema de *scripting* que facilite la tarea de desarrollar el controlador del personaje principal y, más adelante, de los enemigos.

Scripts genéricos para controlar personajes y monitorizar sus estadísticas

Antes de empezar a desarrollar el controlador del jugador se han programado dos componentes genéricos, que, tal y como se muestra en el apartado *Arquitectura del juego*, implementan funcionalidades que tanto él protagonista como los enemigos comparten.

El componente ControladorPersonaje

En este componente se definen las variables y funciones generales de todos los personajes. En cuanto a las variables tenemos:

- *EstadoPersonaje,* que define la acción que está haciendo el personaje en un momento determinado, para así poder reproducir la animación correspondiente.
- *PartesCuerpo* es un *array c*on los *sprites* que componen el cuerpo de un personaje.
- *Velocidad movimiento, CadenaAtaqueMaxima, CadenaAtaque, VelocidadHorizontal y VelocidadVertical,* variables que definen las características del personaje relativas al movimiento.

Por otro lado, los métodos que podemos encontrar en este componente son:

- La función *ActualizarAnimador*, que se encarga de comunicarle al componente de animación que animaciones reproducir. Esto lo hace dándole valores a variables definidas en el animador, y este, atendiendo a esos valores, reproduce una u otra animación.
- El método *CalcularOrientación*, que, a partir de las velocidades tanto horizontal como vertical del personaje determina hacía que dirección está orientado. Para ello, basta con calcular el arco tangente de las velocidades. Una vez tenemos el ángulo, hay que dividir entre cuatro para saber en qué dirección cardinal está mirando el personaje, teniendo en cuenta que, para simplificar los cálculos no se contemplan ángulos negativos ni mayores a 360º.

Dirección	Ángulo
Este o derecha	Entre 0º y 45º y entre 316º y 359º
Norte o arriba	Entre 46º y 135º
Oeste o izquierda	Entre 136º y 225º
Sur o abajo	Entre 226º y 315º

Tabla 13 Relación de ángulos con direcciones

- La función *OrganizarSprites* se encarga de cambiar de manera dinámica el orden en el eje Z de los *sprites*, de esta forma, los *sprites* de aquellos objetos que deberían estar detrás de otro aparecerán de esta forma. En realidad, el orden en el eje Z no es otra cosa que el orden de dibujado. El juego renderizará antes aquellos *sprites* con orden menor. El orden está hecho a mano, ya que no hay otra forma de programarlo para que el juego sepa cuando el jugador está, por ejemplo, mirando hacia la izquierda, por lo que el objeto de su mano derecha tendría que estar por encima de lo demás.

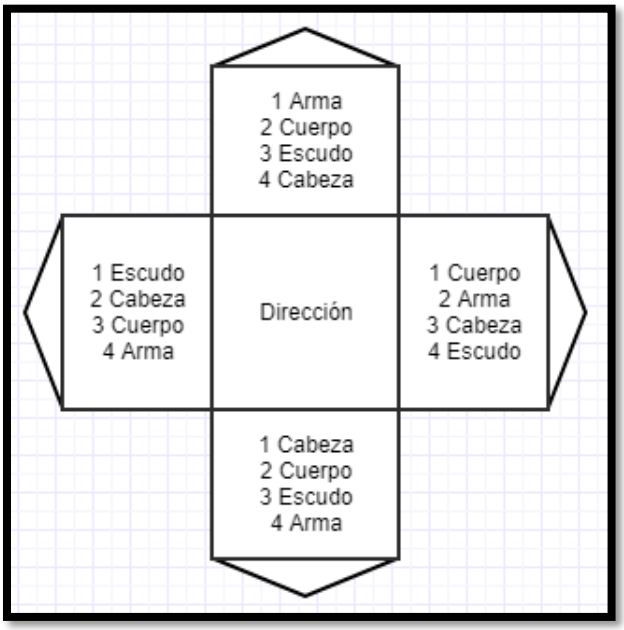


Figura 83 Diagrama de ordenación de sprites atendiendo a su dirección

Fuente: elaboración propia

Ya que todos aquellos objetos que tengan *sprites* que ordenar van a tener cuatro componentes, no hace falta extender más esta función. Si algún objeto no tiene estas cuatro componentes, entonces no necesita ordenar nada, por lo que, en este caso, basta con no ejecutarla.

El componente EstadisticasPersonaje

Este componente se encarga de manejar las estadísticas de cada personaje como son la vida, la energía o el ataque. Además, cuenta con métodos virtuales como *RecibirDaño* y *Morir*, que se implementan en las clases que heredan de esta ya que tendrán comportamientos diferentes. Cada una de estas estadísticas representa una faceta del combate diferente:

- *VidaMaxima* es la salud total de la que dispone el personaje.
- *VidaActual* es la vida actual que tiene el personaje. Si esta llega a cero, el jugador morirá.
- *Fuerza* representa con cuanta fuerza empuja el personaje a aquello a que golpea. A mayor fuerza, más lejos llegará.
- *Ataque* indica el daño que hará el personaje al atacar.
- *RangoAtaque* es la distancia a la que llegan los ataques del personaje. A más rango de ataque, más alcance tendrán los ataques.
- *Armadura* representa la armadura del personaje. La armadura atenúa el daño recibido.

Daño recibido = daño causado – armadura

Cada estadística es un objeto de la clase *Estadística* definida por los siguientes atributos.

- Un entero que representa el valor base
- Una lista de enteros que serán los modificadores.

Este sistema está ideado así para que cuando el jugador se equipe un objeto, base con añadir el modificador correspondiente de dicho objeto, o sustraerlo si se elimina.

Cuenta además con tres métodos para gestionar estos atributos. Dos de estos métodos se encargan de añadir modificadores o de eliminarlo y el tercero y último se encarga de calcular el valor total del atributo.

$$Valor\ de\ atributo = Valor\ base + \sum Modificadores$$

Implementación del personaje protagonista

Para empezar a crear a nuestro personaje jugable, empezaremos creando un objeto del juego al que le añadiremos los componentes. Una vez creado, nuestro objeto Jugador tendrá la siguiente estructura:

Figura 84 Estructura del jugador

Fuente: elaboración propia

- Los objetos *Head*, *Chest*, *Weapon* y *OffHand* son los sprites mediante los cuales se representará al jugador en el mundo.
- *Shadow* es la sombra del jugador y estará siempre a los pies de este.
- *Player Light* es la luz que acompañará al protagonista. Por motivos de jugabilidad y optimización, estará siempre sobre él.
- *InteractUI* y *BurningParticle* son objetos que cumplen funciones específicas en momento determinados, por lo que la mayor parte del tiempo están desactivados.

La razón por la cual se ha seguido esta estructura es que, en la escena de un videojuego, los objetos siguen una jerarquía de árbol. Unity no es diferente en este aspecto. Por eso, en lugar de hacer cada objeto por separado y aplicar la lógica tantas veces como sea necesario, podemos hacer un solo objeto *Player*, con toda la lógica del control del jugador, por lo que esta se aplicaría y ejecutaría una sola vez por ciclo.

Para poder desplazar al jugador por la pantalla necesitaremos aplicarle dos componentes que Unity nos proporciona: *BoxCollider2D* y *RigidBody2D*.

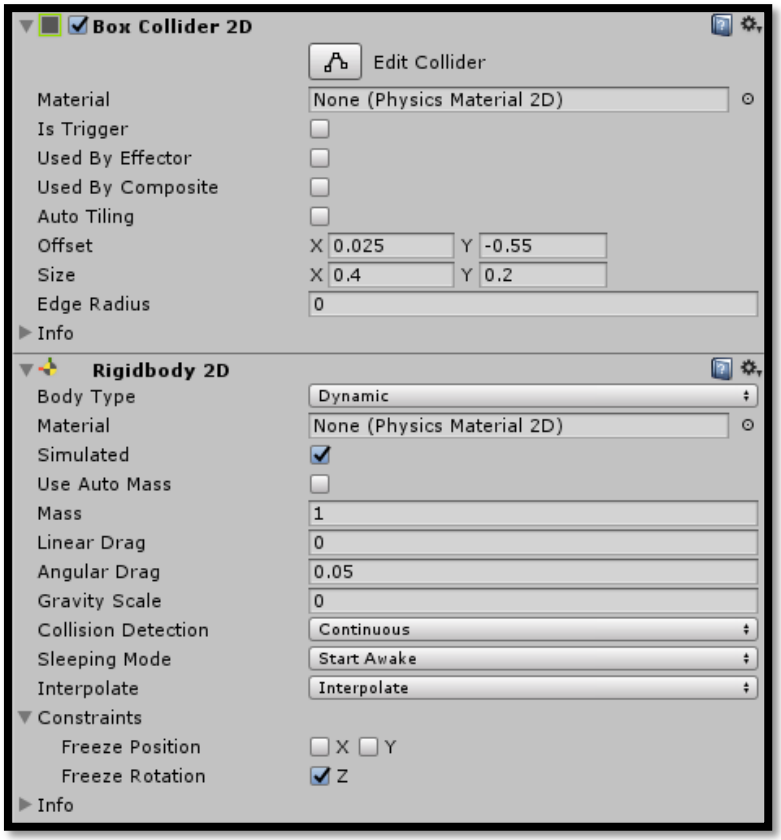


Figura 85 Configuración de las físicas del jugador

Fuente: elaboración propia

Estos dos componentes suelen ir juntos, ya que su combinación permite controlar objetos con físicas.

- *BoxCollider2D* define los límites del espacio que el objeto ocupa en el mundo físico.
- *RigidBody2D* define como las físicas se aplican al cuerpo.

Una vez configurados estos componentes, podremos visualizar en la pantalla de la escena de Unity el espacio que ocupará el jugador, es decir, sus colisiones con el mundo.

Figura 86 Colisiones del jugador

Fuente: elaboración propia

La lógica del jugador

En un juego en el que el jugador siempre controla al mismo personaje, el controlador de este es el aspecto más importante, ya que será el único medio de comunicación entre el jugador y el mundo del juego

Pero, antes de hablar del controlador del jugador, se definirán las diferencias en los métodos *RecibirDaño* y *Morir* de sus estadísticas. En cuanto a *RecibirDaño*, se caracteriza en el jugador por tener en cuenta si está bloqueando con el escudo o no. Para el método de *Morir*, si el jugador muere, terminará la partida.

En cuanto al controlador del jugador, será una máquina de estados básica:

- Tendrá variables que serán los estados, para evitar y crear y destruir objetos dinámicamente, además del estado principal activo.
- Tendrá un método para cambiar de estados
- Ejecutará el estado una vez cada ciclo de ejecución.

Todos los estados del jugador heredan de la clase *EstadoJugador*. Esta clase tiene tres métodos sin definir, ya que cada estado que herede de *EstadoJugador* actuará de manera diferente:

- *Ejecutar:* esta función reemplaza a la función *Update* ya que esta está reservada para Unity y no queremos que todos nuestros estados se ejecuten a la vez, solo uno de ellos. *Ejecutar* contendrá la lógica de cada estado.

- *RecogerEntrada:* esta función se encargará de gestionar la entrada, como pulsaciones de teclas o botones. No todos los estados se comportarán igual ni permitirán cambiar a otros estados específicos.
- *ReiniciarEstado*: esta función se encargará de reiniciar el estado cada vez que se va a utilizar. No haría falta esta función si los estados se destruyesen y creasen, pero esto sería una tarea muy pesada en cuanto a computación.

Volviendo al controlador del jugador, nos encontramos con las siguientes funciones:

- *Update*, que es una función predefinida y reservada por Unity. Todos los componentes la pueden definir y se ejecuta una vez cada ciclo de ejecución, es decir, si un sistema opera a 60 FPS (*Frames Per Second* o Imágenes Por Segundo), la función *Update* se ejecutará 60 veces en un segundo. En el controlador del jugador, *Update* ejecutará el estado actual y actualizará los valores de los animadores del jugador para asegurarnos que se reproducen las animaciones pertinentes.
- La función *CambiarEstado* se encarga de cambiar el estado actual por el estado deseado y reiniciar este último.

Ahora, se detallarán cada uno de los estados en los que el jugador puede encontrarse:

- **Reposo**: el jugador no está haciendo nada. Cualquier entrada causará que el jugador tome una acción, abandonando este estado.
- **Correr**: el personaje se desplazará por la pantalla en las direcciones que el jugador le indique.

- **Esprintar**: el jugador podrá hacer que el personaje aceleré a gran velocidad durante un breve periodo de tiempo, esquivando enemigos y dejando una nube de polvo a su paso.

Figura 87 El protagonista esprintando

Fuente: elaboración propia

- **Atacar**: El jugador podrá ejecutar una serie de ataques. La cantidad de ataques en la cadena dependerá del arma que utilice. En adición a esto, el jugador podrá bloquear y esprintar en entre los ataques de una cadena e inmediatamente después retomarla. En cuanto a la implementación, los ataques, tanto del jugador como de enemigos, se comprueban mediante una técnica de trazado de rayos, para aliviar carga de computación gracias, especialmente, al uso de capas de colisiones. Estas capas actúan como agrupaciones de objetos del mismo tipo, por ejemplo: todos los enemigos en una misma capa, el jugador en una capa específica, etc. De esta forma, solo hay que comprobar si el rayo golpea algún objeto en la capa deseada.

Figura 88 Diferentes ataques del jugador

Fuente: elaboración propia

- **Bloquear con escudo**: el jugador podrá bloquear ataques con su escudo. El bloqueo tiene tres fases:
    - **Movimiento del escudo**: durante esta fase, el personaje estará colocando el escudo delante suyo, y por lo tanto no podrá bloquear con el aún.
    - **Bloqueo perfecto**: por un momento, el escudo del jugador se iluminará. Cualquier bloqueo realizado durante este periodo será más efectivo.
    - **Aguantar la posición**: el jugador podrá mantener el botón para mantener el escudo levantado, bloqueando así cualquier ataque, pero con una eficacia reducida.

Figura 89 Fases del bloqueo con escudo

Fuente: elaboración propia

- **Beber poción**: el jugador podrá beber una poción para recuperar algo de salud. Cualquier daño recibido mientras bebe cancelará el efecto y consumirá la poción.

Figura 90 El jugador bebiendo una poción

Fuente: elaboración propia

- **Transición entre habitaciones**: el personaje se moverá de una habitación a otra. Una vez llegue a su destino, el jugador podrá controlarlo de nuevo.
- **Morir**: cuando la salud del personaje llegue a cero, este morirá y dará paso a la pantalla de las estadísticas de la partida.

Figura 91 Un jugador muerto

Fuente: elaboración propia

Problemas encontrados y soluciones aplicadas

Durante esta iteración el mayor problema ha sido que los *sprites* del jugador eran insuficientes para la cantidad de estados que había que implementar. Como solución a este problema, se han dibujado el resto de *sprites*, y, aunque esto haya causado un ligero retraso en esta iteración, es un problema que no habrá más adelante, cuando haya tareas más importantes que realizar.

Figura 92 Spritesheet inicial del jugador

Fuente: elaboración propia

Figura 93 Spritesheet del jugador en su versión final

Fuente: elaboración propia

Podemos apreciar que, después de esta iteración, la *spritesheet* del jugador cuenta con treinta *sprites* más.

Segunda iteración

El objetivo principal de esta iteración era implementar los enemigos y su inteligenciar artificial, además de diseñar el sistema de interacciones.

Implementación de los enemigos

La primera versión de los enemigos ha sido muy sencilla de implementar, ya que los componentes genéricos que se diseñaron en la iteración anterior han ayudado mucho a la hora de ahorrar tiempo y simplificar el trabajo.

Los enemigos de *Into the Crypt* tienen un comportamiento general:

1. El enemigo deambula por el mapa, descansando en intervalos de tiempo variable.
2. Si el enemigo ve al jugador, lo perseguirá con el fin de acercarse.
3. Si llega a alcanzar al jugador, intentará atacarlo. Los enemigos tienen un rango de ataque, es decir, una distancia a la que hacen daño si atacan. Un enemigo considera estar cerca del jugador cuando la distancia entre estos es menor al rango de ataque del enemigo.
4. Al terminar un ataque, el enemigo retomará el aliento durante un breve momento y reanudará su ciclo de comportamiento.

El componente *ControladorEnemigo*

Un enemigo se define por dos componentes principales, igual que el jugador. Estos componentes son un heredero de *ControladorEnemigo* y *EstadisticasEnemigo*. Ya que hay diferentes comportamientos en los enemigos, el controlador de cada uno de estos tipos será diferente, aunque compartan algunas funciones como comprobar si pueden ver al jugador.

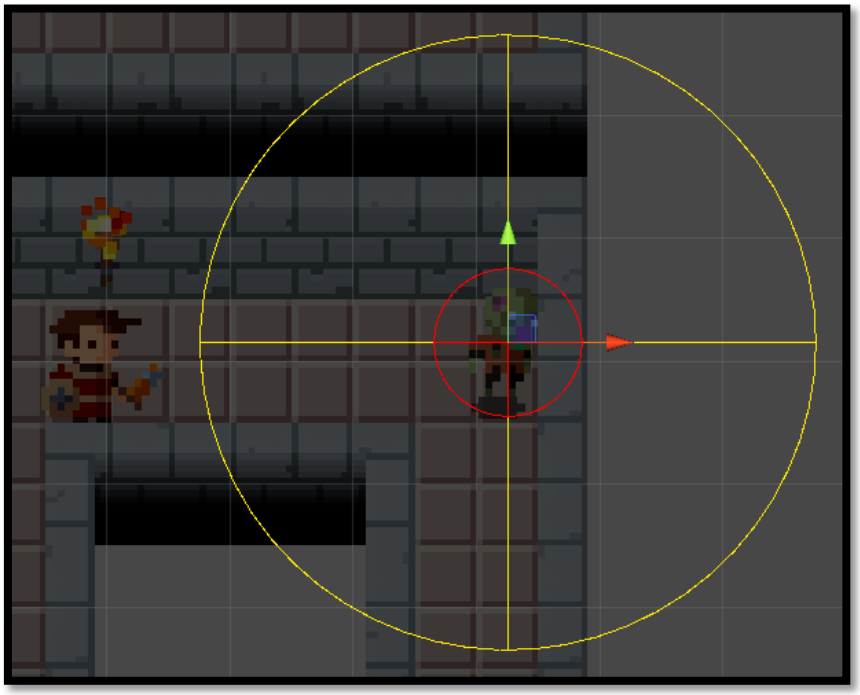


Figura 94 Un enemigo deambulando

Fuente: elaboración propia

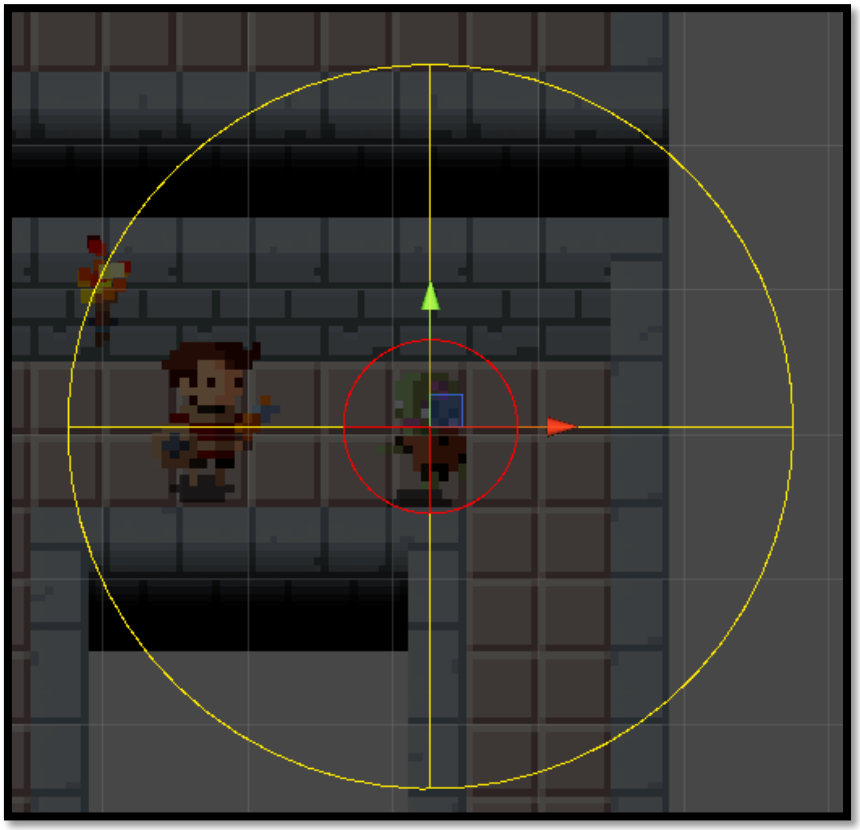


Figura 95 Un enemigo que ha detectado al jugador y lo persigue

Fuente: elaboración propia

La función principal del controlador de enemigos es *ElegirComportamiento*, en la que, mediante el uso de un *Behaviour Tree* o árbol de comportamiento, elige que acción realizará a continuación, basándose en un conjunto de condiciones. El funcionamiento de estos se explicó en la sección *Arquitectura del juego*. Aunque en el juego hay una moderada cantidad de árboles distintos (que se corresponden a los distintos patrones de comportamiento de los enemigos), cuyas implementaciones son extensas, hay formas de optimizarlos y ahorrar trabajo de implementación.

Normalmente, los enemigos se clasifican atendiendo a sus métodos de ataque. Las principales clasificaciones son:

- **Golpeador**: deambulará hasta encontrarse con el jugador, tras lo cual se acercará para golpearle.
- **Vagabundo**: el enemigo deambulará todo el tiempo, haciendo caso omiso del jugador. Si ambos entrasen en contacto, el jugador recibiría daño.

- **Tirador**: el enemigo deambulará hasta encontrar al jugador. Una vez detectado, le disparará desde una distancia segura.

En *Into the Crypt*, los enemigos se comportan de formas similares a estas, aunque en ocasiones se añaden nuevas acciones, que representan las habilidades de cada enemigo para aumentar la diversidad entre enemigos.

Reduciendo la tarea de implementación: el rango de ataque

Implementar la inteligencia artificial de los enemigos, especialmente cuando hay varios tipos, es muy costosa ya que conlleva, además de la implementación, las pruebas, el arreglo de errores y las optimizaciones pertinentes, el diseño del árbol. Por eso, y para minimizar la cantidad de trabajo, particularmente a la hora de implementar golpeadores y tiradores, se ha jugado con el rango de ataque de los enemigos, ya que, fundamentalmente, un enemigo que ataca a distancia es, en esencia, un enemigo que ataca cuerpo a cuerpo con mucho más rango de ataque. Por esto, el árbol de comportamiento de ambos tipos de enemigos es el mismo, aunque cambien sus métodos de ataques.

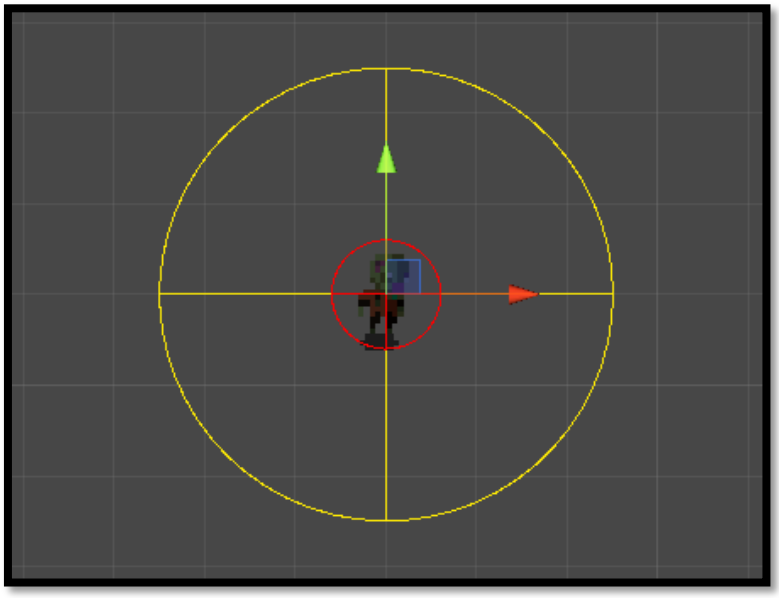


Figura 96 Un enemigo cuerpo a cuerpo, con un radio de visión menor (circulo amarillo) y un rango de ataque aún menor (círculo rojo).

Fuente: elaboración propia

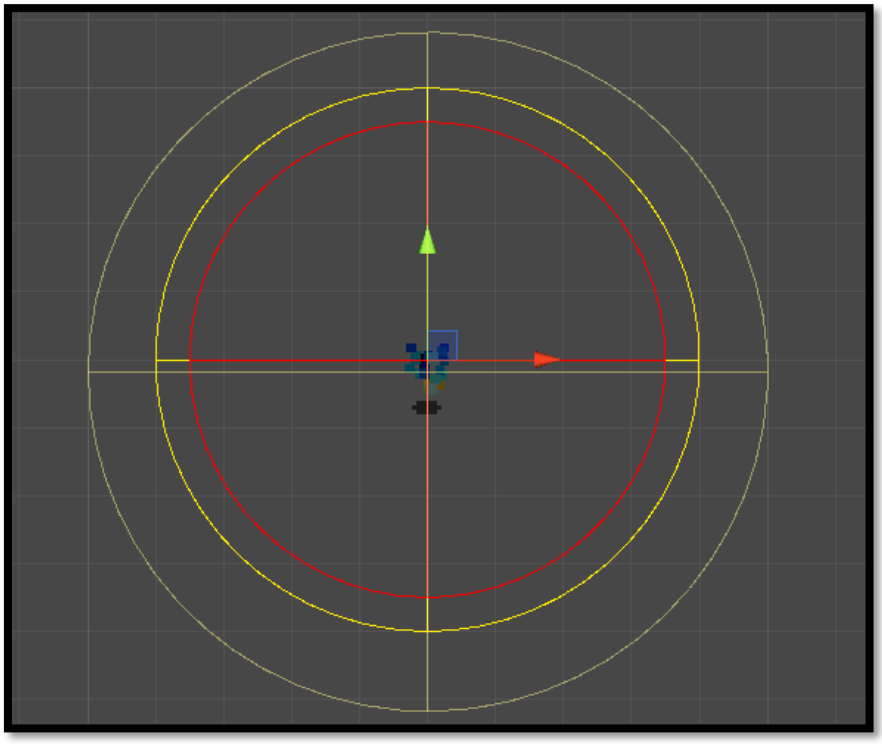


Figura 97 Un enemigo tirador, con un radio de visión alto (circulo amarillo) y un rango de ataque amplio (círculo rojo).

Fuente: elaboración propia

El componente EstadisticasEnemigo

En la sección *La lógica del jugador* se mencionó como las estadísticas del jugador y las de los enemigos son iguales, pero funcionan de manera distintas. Además, como se ha visto en la sección anterior, los enemigos tienen un rango de visión, estadística de la cual el jugador no dispone. Finalmente, la implementación de las funciones *RecibirDaño* y *Morir* es distinta a las del jugador:

- Al recibir daño, los enemigos no bloquean, por lo que todo el daño recibido, atenuado por su armadura, es la cantidad de salud que pierden.
- Al morir, los enemigos dejan caer su botín y otorgan experiencia al jugador.

Diseño del sistema de interacciones

Para preparar la implementación definitiva de los objetos con los que el jugador podrá interactuar se ha desarrollado un componente genérico del que heredarán los demás objetos que permitan interacción como cofres o antorchas.

A este componente se le ha dado el nombre de *Interactuable* y tiene un método virtual *Interactuar*, que implementarán de formas distintas los componentes que hereden de *Interactuable*. Tiene también un método para desactivarse, ya que, a priori, solo se podrá interactuar con los objetos una vez.

Así, una vez el jugador haya interactuado con un objeto, el componente de interacción se desactivará. De esta forma nos aseguramos de que el juego no ejecuta lógica ni cálculos de físicas innecesarios.

Problemas encontrados y soluciones aplicadas

En esta iteración no han surgido problemas debido a la sencillez de las tareas y que partíamos con una buena base en la forma de los componentes *ControladorPersonaje* y *EstadisticasPersonaje*, los cuales han reducido en gran medida el trabajo de desarrollar a los enemigos.

Tercera iteración

Los objetivos principales de esta iteración son: terminar de implementar los objetos interactuables, implementar el sistema de equipamiento e implementar la interfaz y los menús. Además, ya que se van a implementar objetos con interacciones, y que dentro de este grupo entran los objetos de equipamiento, aprovecharemos para implementar el sistema de equipamiento. Aunque en la anterior iteración se empezó a preparar la base para el sistema de interacciones, probarlas sería muy complicado sin elementos visuales. Por eso, la prioridad de esta iteración será la interfaz y los menús.

Implementación de la interfaz

La interfaz de *Into de Crypt* puede ser clasificada en tres apartados diferentes: la interfaz del juego, las interfaces de pausa y los menús.

Interfaz del juego

La interfaz del juego de *Into the Crypt* se ha diseñado para que muestre claramente la información justa y necesaria sin que sea intrusiva ni interfiera con el juego, con el fin de molestar lo mínimo posible al jugador.

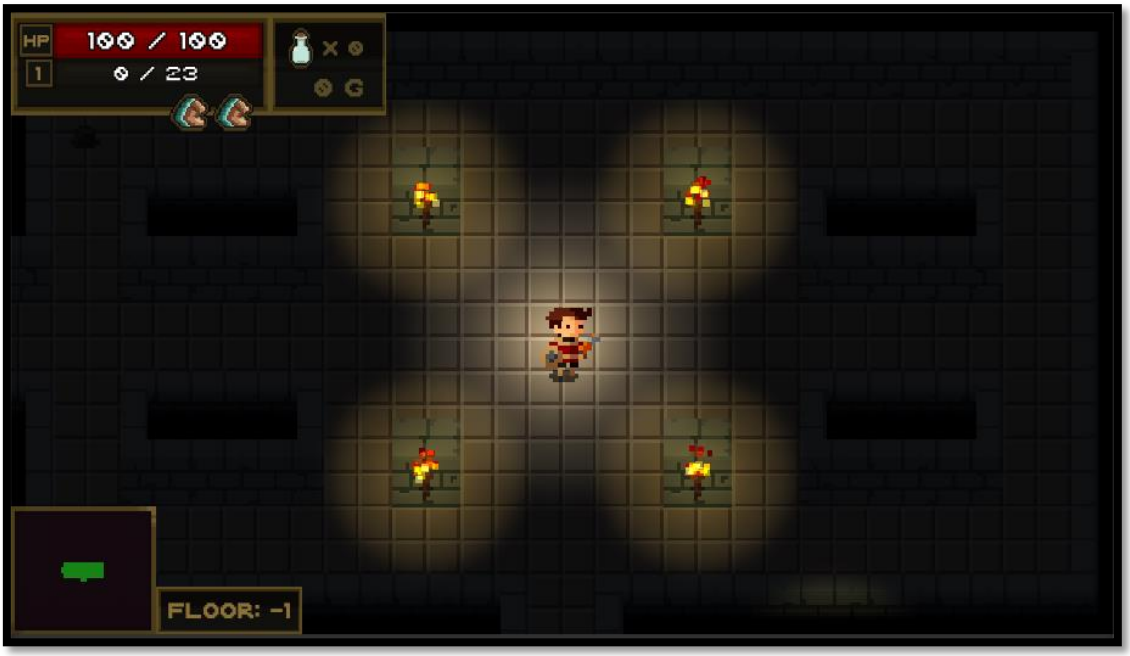


Figura 98 Interfaz del juego

Fuente: elaboración propia

En cuanto al funcionamiento, la interfaz es gestionada a través del componente *UICanvas.* Este componente tiene referencias a todos los objetos de la interfaz del juego dinámicos, es decir, aquellos que cambiarán su valor, texto o apariencia. Para cambiar la información mostrada en pantalla, *UICanvas* tiene métodos para cada elemento (vida, energía, dinero, etc.).

Para actualizar la interfaz podríamos llamar a estos métodos en el *Update,* aunque eso significaría ejecutar estos cambios sesenta veces cada segundo, afectando seriamente al rendimiento del juego. Por eso, se hace uso de delegados. Un delegado es una referencia a un fragmento de código que no se conoce hasta el momento de la ejecución, ya que el fragmento de código puede o no existir, y puede o no referenciarse. Por eso antes de llamar a un delegado hay que asegurarse que existe, es decir, que no es *null.* Cada vez que la información de la interfaz cambia se invoca al delegado pertinente, y este ejecuta la función necesaria para actualizar la interfaz para representar la nueva información. Si por ejemplo cambia la energía del jugador, invocaríamos al delegado de la siguiente forma:

```
if (onEnergyChanged != null) {
     onEnergyChanged.Invoke(currentEnergy, maxEnergy);
}
```

De esta forma, la interfaz solo cambia cuando los datos a representar cambian, reduciendo la carga de ejecución del juego.

Interfaces de Pausa

En *Into the Crypt* existen diferentes formas de pausar el juego:

- Mediante la propia función de pausa.
- Abriendo el inventario.
- Abriendo el mapa.

Cada vez que el jugador hace una de estas acciones, el juego se pausa. Al deshacerlas, el juego continúa su curso normal. Ya que hay varias formas de pausar el juego, se ha desarrollado un componente general, *UIManager*, que se encarga de activar la forma de pausado deseada y desactivar los demás, es decir, si el jugador abre el inventario, este controlador general solo le permitirá reanudar el juego cerrando el inventario, el jugador no podrá abrir el mapa ni pausar el juego.

Cabe destacar que todos los menús, tanto de pausa como del juego, son totalmente usables independientemente de que periférico se use para jugar o navegarlos. Un jugador que decida navegar por los menús con el ratón vivirá la misma experiencia que uno que lo haga con teclado o con mando.

Figura 99 Interfaz del menú de pausa

Fuente: elaboración propia

Desde el menú de pausa el jugador podrá reanudar el juego, volver al menú principal o cerrar el juego.

Figura 100 Interfaz del mapa

Fuente: elaboración propia

Cuando el jugador abra el mapa podrá ver en color blanco todas las habitaciones ya visitadas, en verde la habitación en la que se encuentra y si la ha visitado y abandonado, en amarillo la habitación final del nivel.

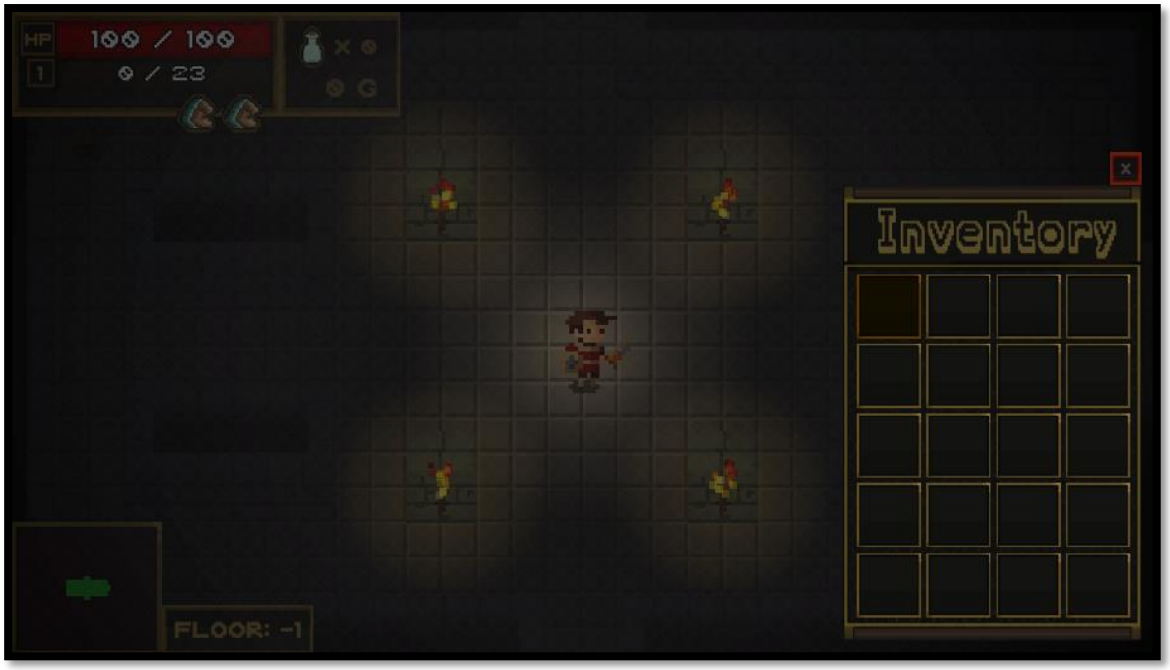


Figura 101 Interfaz del inventario

Fuente: elaboración propia

Desde el inventario el jugador podrá equipar y descartar objetos que haya encontrado. La funcionalidad del inventario se implementará más adelante.

Menús

En *Into the Crypt* hay dos menús separados del juego: el menú principal y el menú final. Estos menús son escenas aparte, pero, en el menú principal hay un total de tres submenús que se intercambian cada vez que se desea pasar de uno a otro, ya que la carga de escenas y la liberación de memoria al abandonar una son tareas pesadas que pueden ralentizar el ordenador.

Los tres submenús del menú principal son:

- **El menú general**: desde aquí, el jugador podrá ir al submenú de jugar o al submenú de opciones, así como salir del juego.

Figura 102 Menú general

Fuente: elaboración propia

- **El menú de juego**: en este menú se presentan al jugador tres opciones: saltarse el tutorial, jugarlo, o volver al anterior menú, es decir, al general.

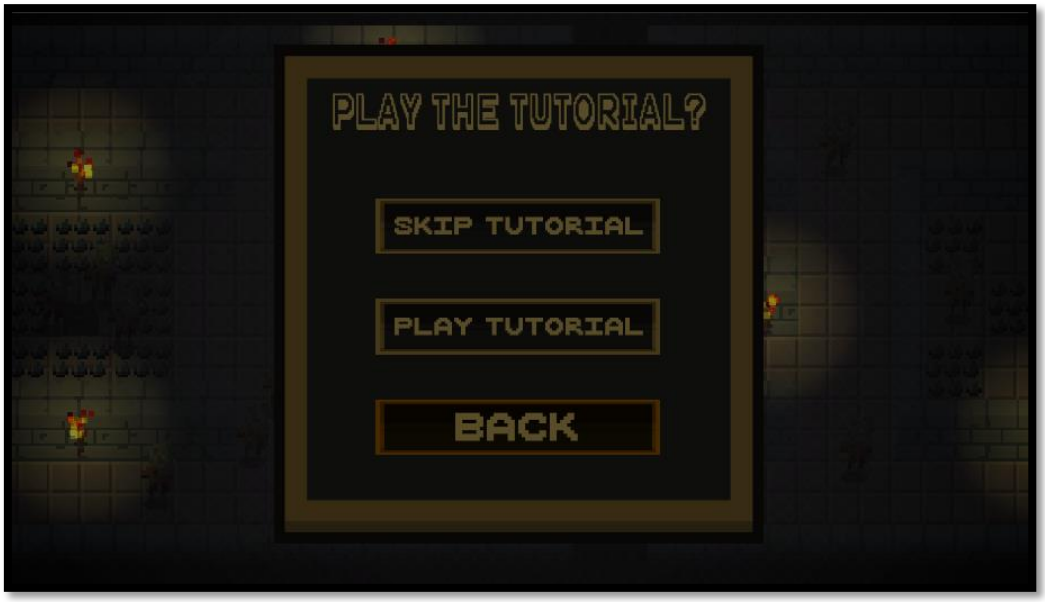


Figura 103 Menú de juego

Fuente: elaboración propia

- **El menú de opciones**: en este menú, el jugador podrá editar algunas características técnicas del juego para que se adapte mejor a su sistema o preferencia a la hora de jugar.

Figura 104 Menú de opciones

Fuente: elaboración propia

  - La resolución deseada puede ser escogida de entre una lista de resoluciones. Esta lista se genera automáticamente al iniciar el programa del juego, ya que solo aparecerán en la lista aquellas resoluciones soportadas por la pantalla del sistema del jugador.

- La calidad viene en tres niveles diferentes: baja, media y alta. A mayor calidad, mejor definición tendrán los *sprites*, mejor *antialiasing*, mejor calidad de luces, etc.
- La pantalla completa permite cambiar entre pantalla completa o modo ventana.
- Silenciar música permite silenciar todo el sonido del juego.
- Si el jugador no quiere silenciar todo el sonido del juego, podrá cambiar el volumen de este.

Implementación de las interacciones

En *Into the Crypt*, el jugador podrá interactuar con múltiples objetos en el mundo de diversos tipos. Cada tipo de objeto tendrá una interacción diferente. Los objetos que permitirán al jugador interactuar con ellos serán:

- Antorchas, la cuales el jugador podrá encender para alumbrar un área.
- Puertas que el jugador podrá abrir.
- Cofres, que contendrán tesoros que el jugador podrá recoger cuando los abra.
- Escaleras que permitirán al jugador avanzar de nivel cuando interactúe con ellas.
- Objetos de equipamiento que se almacenarán en el inventario cuando se recojan.

Todos estos objetos tendrán un componente que heredará de *Interactuable* y que implementarán a su manera el método *Interactuar*. Como ya se mencionó anteriormente, todos los componentes de control se desactivarán al interactuar con ellos.

Además, se mostrará un pequeño botón que simbolice el botón de interactuar para que el jugador sea consciente de que puede llevar a cabo una interacción.

Figura 105 Interfaz de interacción

Fuente: elaboración propia

Sistema de equipamiento

Cuando hablamos de objetos de equipamiento, hay que diferenciar entre el objeto en sí, que actúa como contenedor de información y el objeto con el que el jugador podrá interactuar, que aparecerá en el mundo.

Información del equipamiento

El contenedor de información de los objetos tiene la siguiente estructura.

- Las variables *ModificadorDaño, ModificadorFuerza, ModificadorArmadura y ModificadorBloqueo* son estadísticas que luego se añaden a la cola de modificadores en las estadísticas del jugador cuando este los equipe.
- La variable *Icono* es el icono con el que se mostrará el objeto cuando este esté en el inventario.
- *HuecoDeEquipo* y *TipoArma* son dos enumeraciones que indican donde se equipan (cabeza, cuerpo, arma, escudo) y que tipo de arma son (espada, hacha, lanza, ninguno). Una enumeración es un alias que el programador da a unos miembros de un tipo de datos específicos, normalmente enteros. Facilitan la programación al aumentar la claridad del código. En el momento de la compilación, el compilador desecha los alias y usa solo los valores de la variable.

Además, los objetos de este tipo heredan de *ScriptableObject*, la cual es una clase especifica de Unity que nos permite crear un submenú dentro del editor para crearlos.

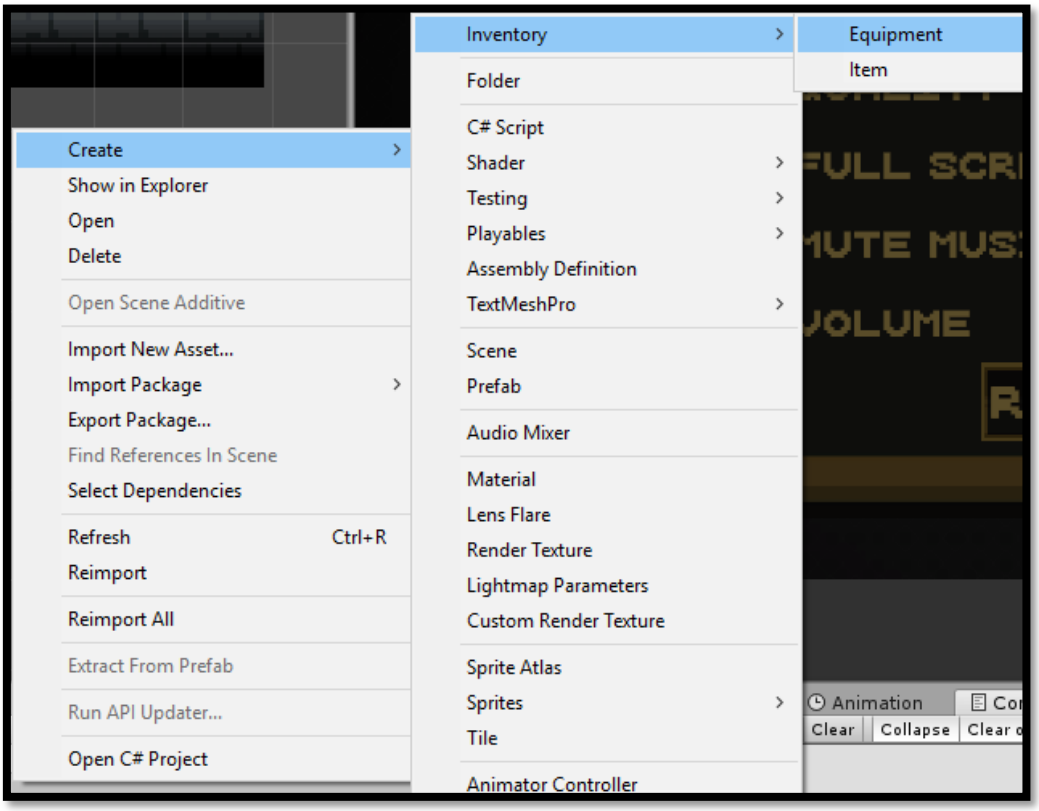


Figura 106 Menú de creación de objetos.

Fuente: elaboración propia

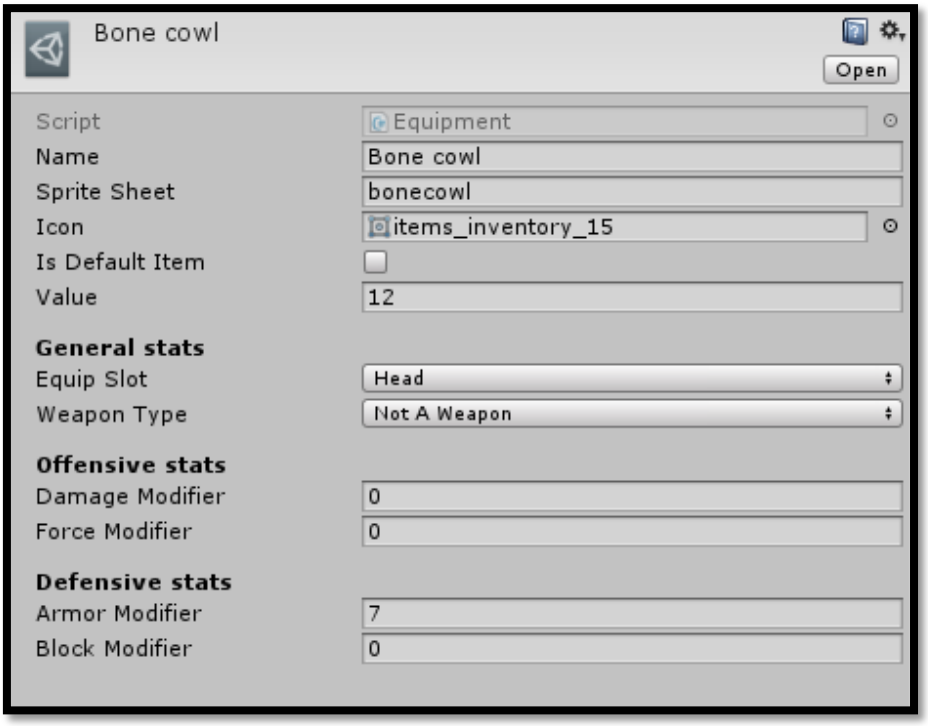


Figura 107 Configuración de un objeto

Fuente: elaboración propia

Objetos en el mundo

Una vez tenemos un objeto creado, es decir, el contenedor de datos preparado, podemos añadirlo a un objeto del mundo con el que podrá interactuar el jugador para recogerlo. Para eso, estos objetos implementan el método *Interactuar* de una forma específica. Al interactuar con un objeto del mundo, intentará añadir el objeto al inventario del jugador. Si no puede, debido a que el inventario está al máximo de su capacidad, no hará nada.

Los objetos tienen, además, un *sprite* cuando están en el mundo y uno más detallado para cuando están en el inventario.

Figura 108 Sprites de los objetos en el mundo

Fuente: elaboración propia

Figura 109 Sprites de los objetos en el inventario

Fuente: elaboración propia

Sistema de inventario

El inventario tiene dos partes en las que nos centraremos: los huecos de inventario y las funciones del controlador que gestionan tanto el inventario como el equipamiento del jugador.

Figura 110 Interfaz del inventario

Fuente: elaboración propia

Los huecos de inventario contienen los objetos que el jugador recoge y componen, además, la interfaz del inventario.

Cuando un jugador recoge un objeto, él se añade al inventario, pero, si el jugador decidiese equipar o descartar dicho objeto, se llamaría al método se eliminaría del inventario. Para equipar un objeto, el jugador puede seleccionarlo con el mando o teclado o hacer clic en el icono del objeto. Por otra parte, para descartar un objeto, el jugador tendrá que pulsar el botón pertinente o hacer clic en el botón de descartar.

Figura 111 Ejemplo de hueco de inventario con un objeto

Fuente: elaboración propia

En la interfaz se muestra, además, un panel con las estadísticas de cada objeto, así como su nombre y el valor monetario de este. La información se extrae del objeto que el hueco de inventario contiene.

Figura 112 Panel con información sobre un objeto

Fuente: elaboración propia

En cuanto a los métodos para gestionar el inventario y el equipamiento, el funcionamiento es muy sencillo, ya que el núcleo de esta funcionalidad apenas son tres métodos.

Los dos primeros métodos, *Equipar* y *Desequipar* se encargan de, como su nombre indica, equipar o desequipar un objeto de la casilla correspondiente del jugador. Estas casillas son cuatro y las representan un valor entero que indica un orden definido mediante la enumeración *HuecoDeEquipo* (Cabeza = 0, pecho = 1, arma = 2, escudo = 3). A la hora de equipar un objeto, si ya había uno antes en esa casilla, lo reemplazará, añadiendo el objeto anterior al inventario.

Cuando un objeto se equipa, se invoca un delegado que se encarga de recalcular las nuevas estadísticas del jugador. El funcionamiento de los delegados se explicó en la sección *Interfaz del juego*

Cuando se equipa un objeto y se invoca con éxito a este delegado, se ejecuta una función (*EquipamientoCambiado*) cuyo funcionamiento es muy sencillo. Recibe como parámetro dos objetos, uno a equipar y otro a desequipar. Si el objeto a desequipar existe, es decir, no es nulo (*null*), quita los modificadores de estadísticas del objeto antiguo. Después, y solo si el objeto a equipar existe, añadirá los modificadores y actualizará el tipo de arma si así fuese necesario. Cabe destacar que, aunque en la gran mayoría de casos se cambie un objeto por otro, existe la posibilidad de equipar un objeto donde antes no lo había y de desequipar un objeto completamente.

Sistema visual de equipamiento

En cuanto al apartado visual del sistema de equipamiento, el funcionamiento de este es el mismo que se indicó en el apartado *Sistemas de equipamiento en juegos como Diablo II o MMORPG*.

En la estructura del objeto del jugador (Figura 83), se ven unos GameObjects, de los cuales cuatro de ellos solo tienen un *SpriteRenderer*, un componente de Unity que se encarga de dibujar *sprites* por pantalla. El *Sprite* que dibuja en cada momento depende del *frame* actual de la animación, y la animación, a su vez, depende del estado del jugador. Todas las animaciones de un personaje componen una *spritesheet*, pero, en el caso de *Into the Crypt*, tendremos una *spritesheet* por cada objeto que el jugador pueda equipar.

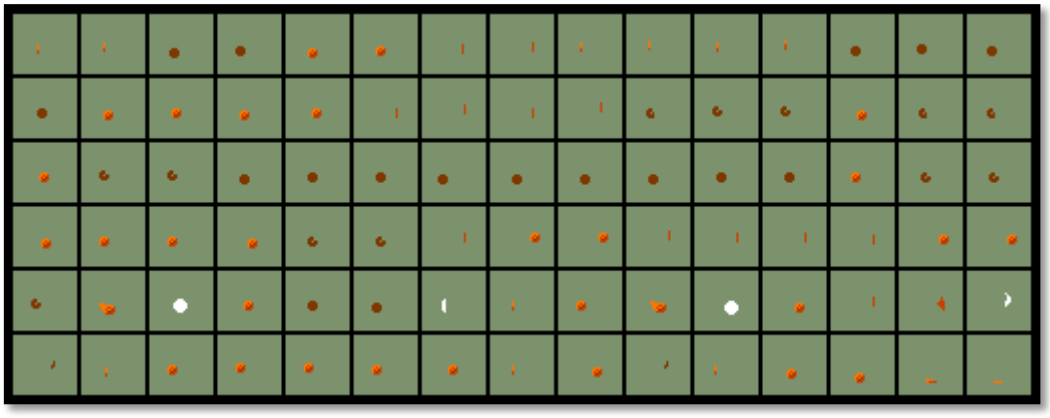


Figura 113 Spritesheet del escudo de bronce

Fuente: elaboración propia

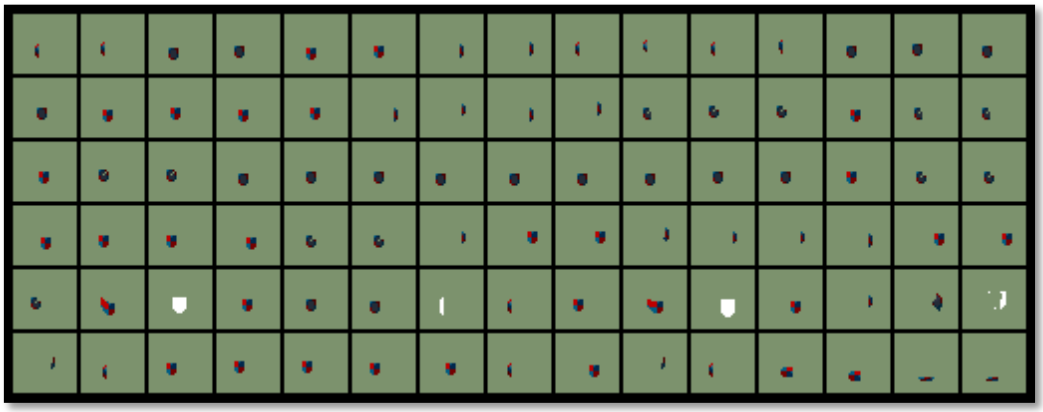


Figura 114 Spritesheet del escudo de caballero

Fuente: elaboración propia

Una vez tenemos todos los *spritesheets* de los objetos, podemos hacer múltiples combinaciones con ellos.

Figura 115 El protagonista de Into the Crypt, llevando un casco de caballero, una armadura de huesos, un hacha de fuego y un escudo de hierro.

Fuente: elaboración propia

Pero ¿Cómo se alcanza este estado de implementación? A continuación se detallará en profundidad, tanto el planteamiento como el desarrollo y se explicará el funcionamiento del sistema a nivel de código.

El gestor de contenidos de Unity

En su versión actual, Unity no nos permite saber que *spritesheet* está renderizando en un momento determinado, ya que al separar las *spritesheets* en *sprites*, Unity los almacena como *sprites* separados. Entonces, ¿cómo cambiamos un *sprite* por otro dependiendo del objeto que lleve equipado el jugador? La respuesta a esta cuestión es el gestor de recursos de Unity, y la solución pasa por un componente que pueda explotar las cualidades de este.

Este componente, *AspectoAnimación*, tiene los siguientes atributos:

- *NombreSpritesheet* : el nombre de la *spritesheet* actual.
- *SubImagenes*: todos los *sprites* de la *spritesheet* actual.
- *Renderizador*: el SpriteRenderer objetivo, aunque solo vaya a ser uno, tiene que estar almacenado en un *array* de una sola posición, ya que la búsqueda del gestor de recursos no permite volcar resultados en algo que no sea un *array*.
- *NombreSprite*: nombre del *sprite* actual.
- *NuevoSprite: sprite* actual.

Los objetos (contenedores de información) tienen también un *string* que es el nombre de la *spritesheet* que les corresponde. Al equiparse, un delegado cambia el nombre de la *spritesheet* de la parte del cuerpo correspondiente por el nuevo de *spritesheet*, haciendo que, además, cargue todos los *sprites* que componen la nueva *spritesheet*. En cuanto a cómo cambia el *sprite* mostrado por pantalla, el funcionamiento es el siguiente.

Esta funcionalidad tiene que ejecutarse en una vez cada *frame*, pero en vez de hacerlo en *Update,* se hará en *LateUpdate*, ya que esta siempre se ejecutará después que *Update,* por lo que todos los cálculos y más importante aún, todos los cambios de objetos y estados que el jugador haya llevado a cabo habrán terminado, evitando así posibles errores. En cuanto a su funcionamiento, busca en la lista de *sprites* aquel *sprite* que corresponda a la *frame* actual de la *spritesheet* actual y los reemplaza en el *SpiteRenderer*

Problemas encontrados y soluciones aplicadas

En esta iteración han surgido dos problemas principales. Uno de ellos era que, en su primera versión, el sistema visual de equipamiento era muy poco óptimo y tras implementarlo, el rendimiento del juego. La solución fue reducir la carga de ejecución del código para optimizar el rendimiento. Esta optimización se detallará más adelante.

El segundo problema consistía en que, en el sistema de equipamiento visual, los nombres de los *sprites* tienen que coincidir, y, ya que Unity llama a los *sprites* por el nombre de la *spritesheet* y el número que ocupa (SpritesheetEjemplo_0, SpriteSheetEjemplo_0), no funcionaba correctamente. La solución a este problema pasa por llamar a todas las *spritesheets* igual y hacer la diferenciación de nombres en las carpetas en la que se encuentran. Por ejemplo: *CascoHierro/spritesheet.png y CascoBronce/spritesheet.png*.

Estas soluciones han sido fáciles de aplicar por lo que no han provocado retrasos en la iteración.

Cuarta iteración

El objetivo principal de esta iteración es programar la generación procedimental de mazmorras.

Implementación del algoritmo de generación de mazmorras

El sistema de generación de mazmorras es, quizá, la parte más desafiante de este proyecto, aun tomando una aproximación sencilla como es la generación de mazmorras al estilo *The Binding Of Isaac*.

El sistema de generación de mazmorras se divide en dos pasos.

La creación del nivel

La primera parte del sistema de generación es la creación de un nivel. De esto se encarga el componente *LevelGeneration*. En él, podemos encontrar, entre otros, los siguientes atributos:

- Un atributo de tipo Vector2 *TamañoMundo* que define el tamaño del mundo en habitaciones, es decir, el valor por defecto es de 4x4 habitaciones.

- Estas medidas del mundo se multiplican luego por dos para dar más libertad de movimiento al algoritmo y para asegurarnos de que existe una habitación central. Estas nuevas dimensiones se almacenan en *TamañoGridX* y *TamañoGridY*.
- El atributo *NumeroHabitaciones que* indica el número de habitaciones que intentará colocar el algoritmo en el mundo.

Una vez tenemos el tamaño del mundo definido, es hora de crear las habitaciones siguiendo los siguientes pasos:

1. El primer paso es colocar la habitación inicial en el centro del mapa y configurarla como tal, redondeado al entero inferior.
2. Luego, partiendo de esta habitación, busca posiciones vecinas en las que poder colocar una habitación, repitiendo esto para cada habitación. Cuanto más lejos está la nueva habitación de la habitación central, más posibilidades tiene el algoritmo de abandonar esa dirección, con el fin de crear mazmorras con varias direcciones.

Para buscar las posiciones, el algoritmo se apoya en dos métodos que se encargan de escoger aleatoriamente en qué posición mirar, podrán decidir si avanzar hacia arriba o abajo o hacia la derecha o izquierda, aunque también pueden decidir no avanzar. La diferencia entre ellos es que uno parte de una habitación cualquiera y otro parte de una habitación determinada cuya cantidad de habitaciones vecinas es uno.

Mientras genera habitaciones, el algoritmo tiene la posibilidad de colocar la habitación final del nivel, aunque, si no fuese así, la última habitación que colocase seria la habitación para avanzar al siguiente nivel.

Ahora, el algoritmo conoce el tamaño de la mazmorra, las habitaciones que tiene y donde se encuentran, pero en el mundo del juego aún no hay nada, ya que todo el proceso hasta este punto ha sido creación y manejo de datos. Para solucionar esto, el algoritmo crea en el mundo los *sprites* que representarán las habitaciones con objetos que posean el componente *SelectorSprite*, que funciona como clase auxiliar que se encarga de cambiar su *sprite* dependiendo de las habitaciones vecinas que tenga, y su color dependiendo del tipo y de si el jugador está en ella en ese momento.

Una vez llegados a este punto, el algoritmo generará un mundo similar al siguiente:

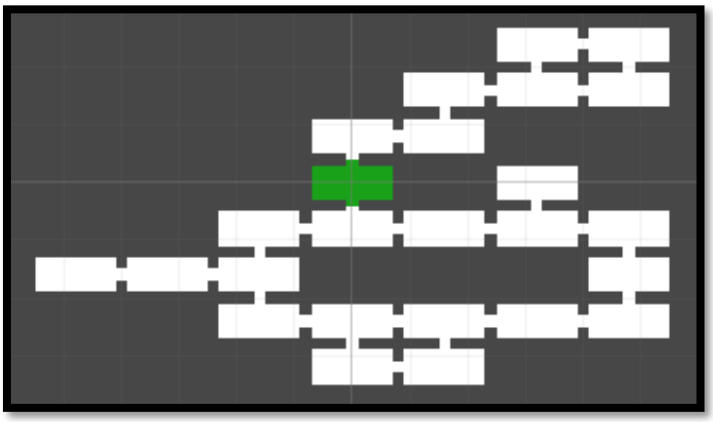


Figura 116 Resultado de la generación de mazmorras inicial con un mundo de 4x4 (8 habitaciones de ancho por 8 habitaciones de alto) y 25 habitaciones

Fuente: elaboración propia

Podemos cambiar tanto el tamaño del mundo y el número de habitaciones para crear mapas más grandes o más reducidos.

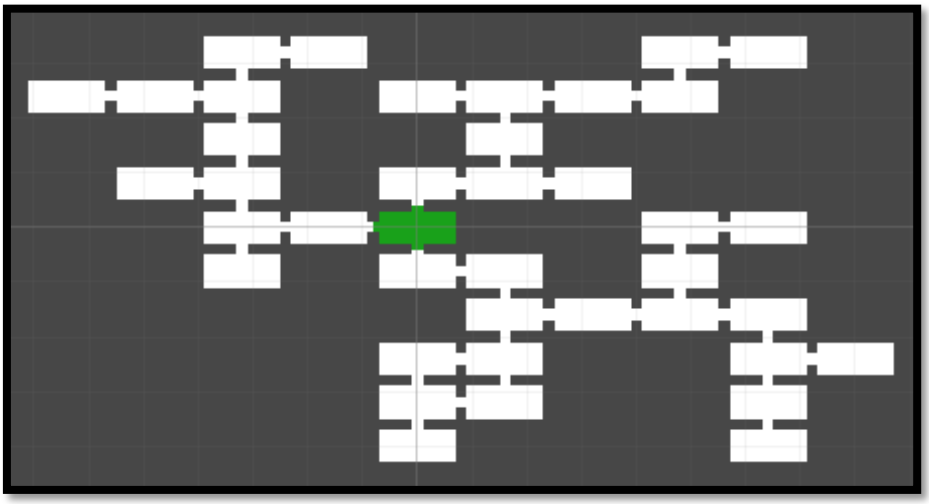


Figura 117 Una mazmorra con un tamaño de 8x8 (16x16) y 40 habitaciones como máximo

Fuente: elaboración propia

Aunque hay que tener cuidado de que el tamaño del mundo y el número máximo de habitaciones concuerde, o podríamos generar mapas deficientes.

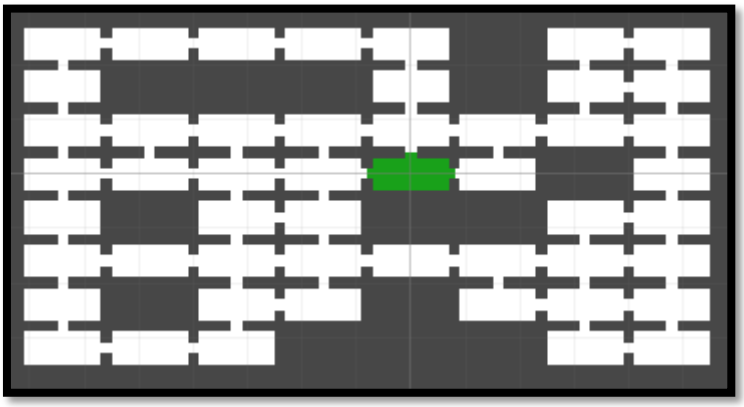


Figura 118 Un mundo generado de manera defectuosa. Tamaño 4x4 (8x8) y 50 habitaciones

Fuente: elaboración propia

Creación del mundo

Al llegar a este paso, el nivel está definido. Tenemos el nivel y las habitaciones, pero no tiene nada de jugable. Para convertir el mapa actual en un nivel jugable, se ha desarrollado un componente, *AsignadorPlantilla*, con un único método que recibe por parámetro el conjunto de habitaciones y para cada una de ellas decide una plantilla que usar de forma aleatoria de entre un total de veinte plantillas. A excepción de la habitación inicial y la final, que tienen plantillas preasignadas.

Estas plantillas son texturas de 17x9 píxeles, en las que cada píxel es de un color diferente, y cada color representa un objeto del juego: enemigos, cofres, paredes, etc. Si cambiásemos el tamaño de las plantillas cambiaríamos también el tamaño de las habitaciones. La razón por la que se han escogido estas dimensiones por tres razones:

1. Las habitaciones son rectangulares y más anchas que altas, adaptándose mejor a las pantallas.
2. Es un tamaño perfecto para que el jugador recorra las habitaciones en el tiempo justo, ni demasiado rápido ni demasiado lento.
3. Son impares, asegurándonos de que las puertas de las habitaciones quedan siempre en el centro de las paredes.

Como se ha mencionado anteriormente, las plantillas de habitación son texturas. Aunque Unity normalmente no permite usar las texturas nada más que para aplicarlas a objetos, se pueden configurarlas para poder usar las texturas con permisos de lectura y escritura.

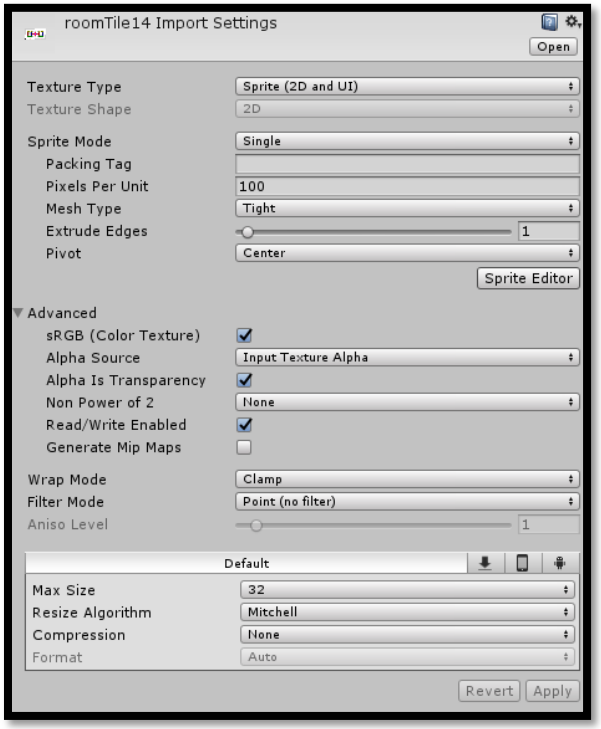


Figura 119 Configuración de una textura para usarla con permisos de lectura y escritura

Fuente: elaboración propia

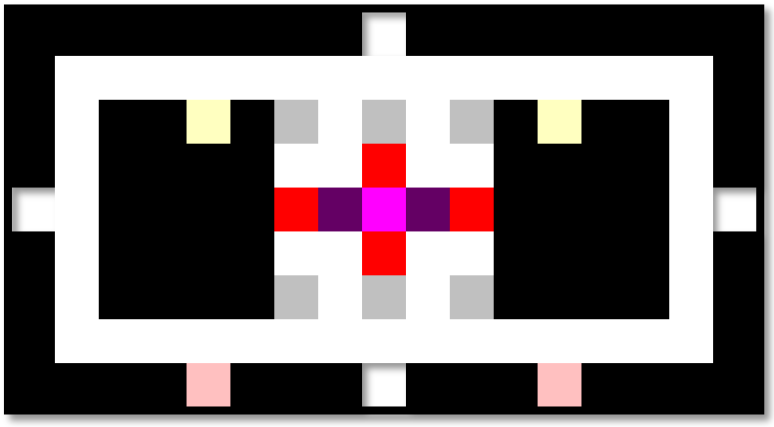


Figura 120 Ejemplo de plantilla de habitación

Fuente: elaboración propia

Color	**Objeto del juego**
	Trampa de pinchos
	Botín
	Suelo
	Enemigo
	Cofre
	Pared
	Puerta
	Pilar
	Antorcha
	Obstáculo
	Trampilla
	Pilar con antorcha encendida
	Poción

Tabla 14 Leyenda de colores para las plantillas de habitaciones

Una vez el algoritmo ha tiene una pareja compuesta por una habitación, que contiene información como la posición, vecino, etc.; y la plantilla, crea un *GameObject* con un componente llamado *InstanciaHabitación*, el cual se encarga de colocar los objetos que su plantilla le indica en el mundo. Este tiene una lista de relaciones entre colores y objetos del juego.

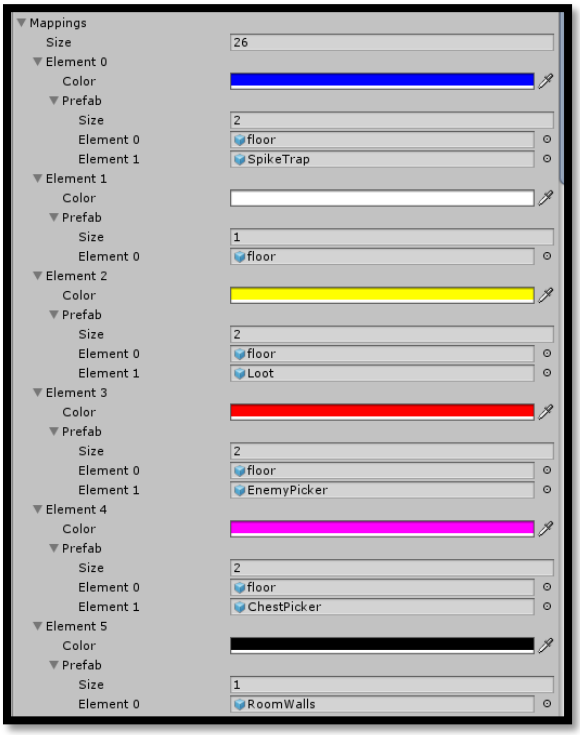


Figura 121 Lista de referencias entre colores y objetos del juego

Fuente: elaboración propia

Posee también un método que se encarga de leer la textura píxel a píxel y comparar el color del píxel con los colores de la lista. Hay que tener cuidado ya que la lectura de texturas se realiza de izquierda a derecha y de abajo hacia arriba, aunque al colocar los objetos del mundo se coloquen de izquierda a derecha y de arriba abajo.

Figura 122 Un nivel generado de manera procedimental completamente jugable

Fuente: elaboración propia

Al ejecutar obtendremos un resultado similar al anterior: un nivel completamente jugable, con una gran variedad de habitaciones diferentes y con múltiples rutas para explorar.

Últimos retoques del algoritmo

Viendo la figura 122 puede parecer que el algoritmo estaría terminado. Sin embargo, en dicha figura se muestra el resultado de la generación de la mazmorra por el bien de este documento. La realidad es que tanto el mapa del primer paso como el mundo del segundo se superponen de la siguiente forma:

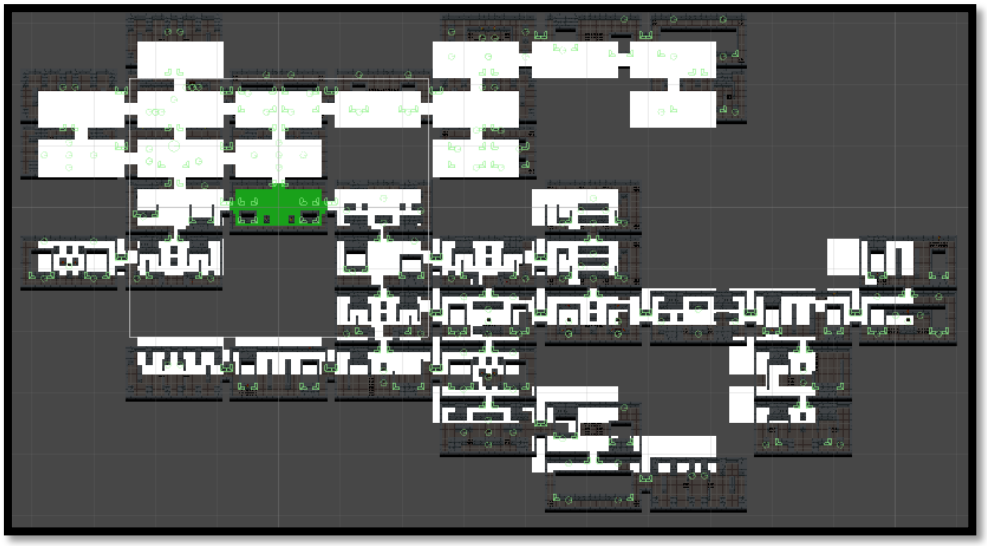


Figura 123 Superposición del mapa y del nivel

Fuente: elaboración propia

Para solucionar esto y sin tener que descartar ningún progreso conseguido a lo largo de la implementación del algoritmo, vamos a crear dos cámaras: una que capte el mundo del juego y lo renderice en el *viewport* principal, es decir, la pantalla y otra que capte el mapa del nivel y lo renderice en una textura que aparecerá en el plano de la interfaz y que actuará como mini mapa. Con esto, conseguimos el siguiente resultado:

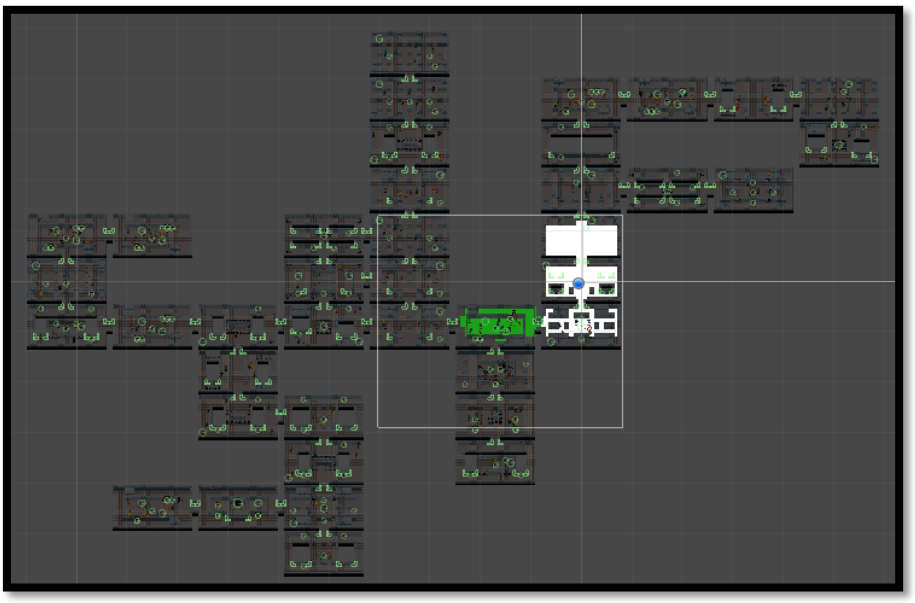


Figura 124 Visión de la escena, con superposición del mapa y del mundo del juego

Fuente: elaboración propia

Figura 125 Pantalla del juego, ningún objeto del mapa se superpone, sino que se renderizan aparte en el mini mapa.

Fuente: elaboración propia

Recordemos que, para la versión final del juego, las habitaciones no exploradas no se verán en el mapa, tal y como se puede apreciar en la figura 125.

Finalmente, hay que hablar del algoritmo de colocación de paredes. Para colocar las paredes juntas y que cada una tenga no solo un aspecto distinto que se corresponda con la sección de la pared que ocupa sino también una colisión distinta, se ha desarrollado un componente que detecta paredes y puertas adyacentes y coloca la pared correcta en su lugar.
El componente *AutoTile* basa su funcionamiento en lanzar rayos mediante *RayCast* en forma de circulo en sus posiciones colindantes, y, si detecta una pared, almacena esa información para al terminar, colocar la pared correspondiente. A continuación, se muestra un ejemplo:

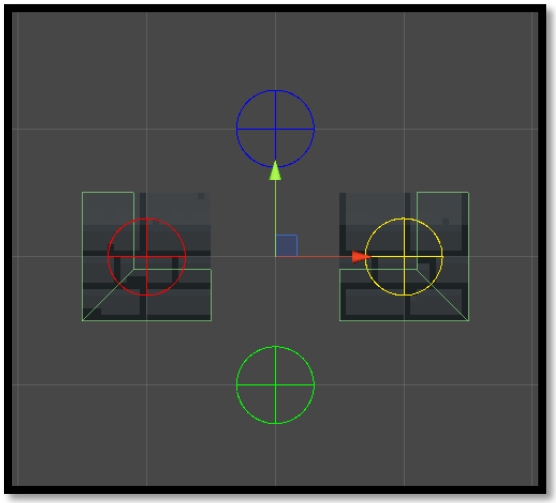


Figura 126 Detección de paredes alrededor de un objeto con un componente AutoTile. El juego se ha pausado para tomar esta captura

Fuente: elaboración propia

Mediante el dibujado de los rayos para depuración que Unity nos brinda, podemos ver en qué posiciones está haciendo la comprobación: azul hacia arriba, rojo a la izquierda, amarillo a la derecha y verde hacia abajo. En este ejemplo preparado, debería detectar que esta entre dos paredes inferiores, y, por lo tanto, debería colocar una pared intermedia inferior.

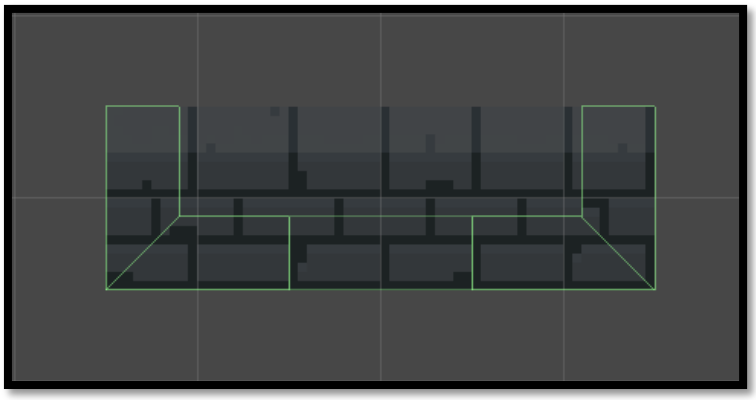


Figura 127 Al reanudar el juego, el algoritmo detecta las paredes correctamente y coloca la pared correspondiente

Fuente: elaboración propia

A partir de Unity 2018 esta tarea es mucho más sencilla ya que viene integrada en Unity, pero, en el momento en el que se desarrolló este algoritmo, Unity aún estaba en su versión 2017, por lo que esta funcionalidad tuvo que ser implementada desde cero por mí.

Problemas encontrados y soluciones aplicadas

Durante esta iteración, y debido a la dificultad del algoritmo a desarrollar, el único problema encontrado ha sido un problema de moral, ya que varios días seguidos batallando contra el mismo error pueden fácilmente minar la más fuerte de las morales. Como solución, decidí tomarme dos días de descanso. Uno lo dediqué a desconectar completamente del proyecto, y otro lo dediqué a ámbitos del juego más triviales y relajados.

Quinta iteración

En esta iteración, el objetivo era terminar todo el contenido del juego (*sprites*, enemigos, etc.) y añadir un sistema de sonido y un tutorial.

Implementación del sistema de sonido

En Unity, si queremos reproducir un sonido, tenemos que añadir a un *GameObject* un componente del tipo *AudioSource*, y, acto seguido, configurarlo añadiéndole un sonido que reproducir, si queremos que se repita, el volumen, el tono, etc.

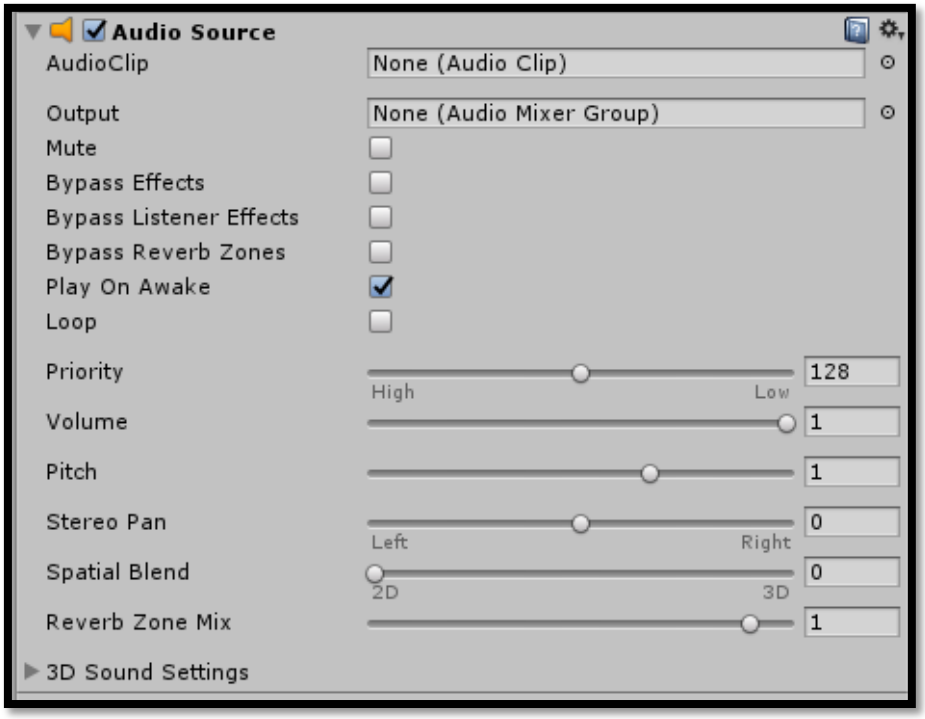


Figura 128 El componente AudioSource

Fuente: elaboración propia

Aunque esto resulta muy intuitivo a la hora de desarrollar, a la hora de mantener un proyecto, gestionar los sonidos se convierte en una tarea muy tediosa, ya que los sonidos están repartidos

entre todos los objetos que tengamos en nuestro juego. Finalmente, habrá objetos que reproduzcan el mismo sonido: un cofre de madera y una puerta podrían reproducir el mismo sonido, por lo que tendríamos dos *AudioSource* reproduciendo el mismo sonido, consumiendo memoria de manera innecesaria.

Como solución al problema del mantenimiento y de la optimización se ha diseñado un sistema de sonido centrado en un solo componente: *ControladorSonido*, apoyado en una clase auxiliar propia *Sonido*.

La clase auxiliar *Sonido*

Aquellos objetos que pertenezcan a la clase *Sound* tendrán las siguientes características:

- *Nombre* es el nombre de este sonido, facilita la gestión por parte del desarrollador y permite realizar una búsqueda basada en el nombre a la hora de reproducir sonidos.
- *Pista* es el sonido o efecto para reproducir.
- *Volumen*, *Tono* y *Repetir* hacen referencia al volumen, tono y si debe repetirse al terminar. Coinciden con variables de un componente *AudioSource*.
- *Source* es el *AudioSource* que corresponde a este audio. Su función se explicará más adelante.

Finalmente, cabe destacar que la clase *Sonido* está definida como *serializable*, es decir, podremos crearla en serie en el editor, facilitando la tarea de desarrollo.

El componente *ControladorSonido*

El componente *ControladorSonido* está definido por dos *arrays* que almacenan los sonidos a reproducir. La diferencia entre ellos radica en el hecho de que uno almacena música para reproducirla de fondo, y otro almacena efectos de sonido.

Estos dos arrays de objetos de tipo *Sonido* son convertidos al empezar el juego en *AudioSource*, añadiéndolos a su mismo *GameObject* y configurándolos con la información que cada *Sonido* almacenaba. De esta forma, todos los *AudioSource* quedan agrupados en un solo *GameObject* y no hay ningún *AudioSource* con sonidos repetidos.

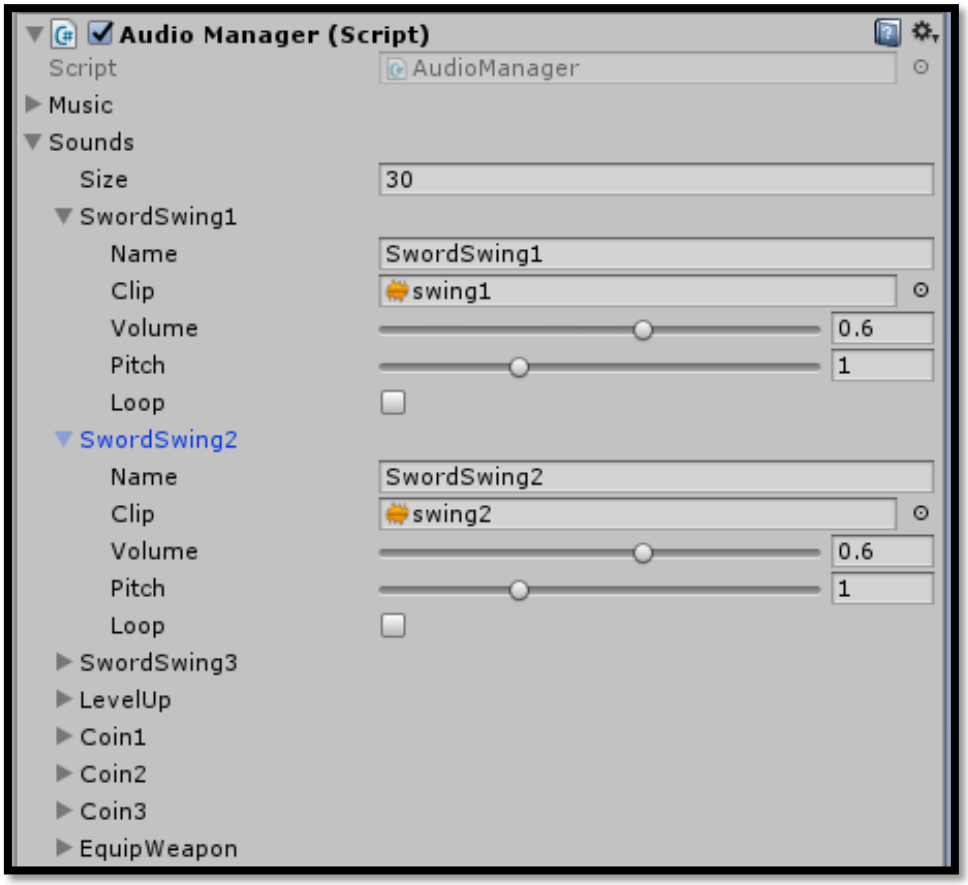


Figura 129 Editor del componente ControladorSonido, con algunos sonidos almacenados

Fuente: elaboración propia

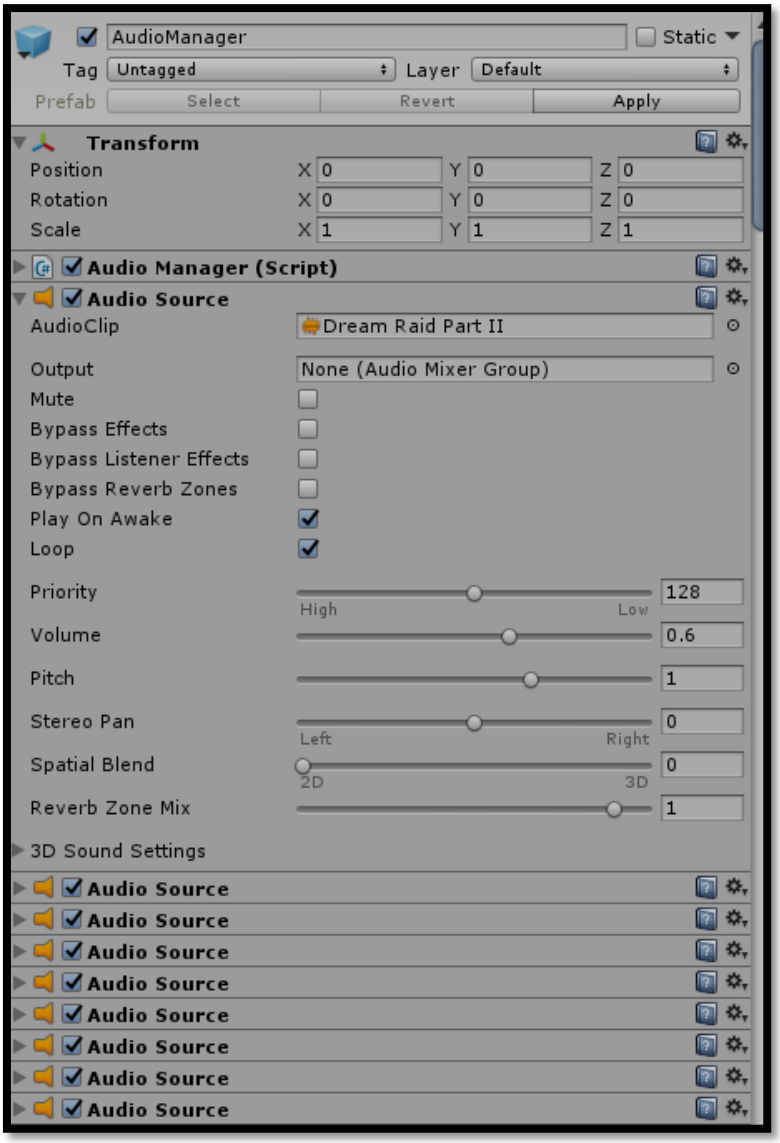


Figura 130 GameObject del ControladorSonido, con todos los AudioSource añadidos

Fuente: elaboración propia

Además, este componente implementa el patrón *Singleton*, por lo que nunca habrá más de un *ControladorSonido*. Además, al escribir la línea siguiente, el objeto será persistente entre niveles, por lo que el *ControladoSonido* que se crea al iniciar el juego, será el mismo para toda la sesión.

```
    DontDestroyOnLoad(this);
```

Como se reproducen los sonidos

Para reproducir sonidos, *ControladorSonido* cuenta con tres métodos:

- Para reproducir música.
- Para reproducir un sonido dado.
- Para reproducir un sonido de una lista.

Los dos primeros tienen un funcionamiento idéntico, a excepción del *array* en el que buscan: uno buscará sonidos y otro buscará música. Estos métodos reciben por parámetro un *string* que representa el nombre del sonido a buscar. El método buscará el sonido y lo reproducirá si existe.

El tercer método recibe un *array* con tantos *string* como se desee. De entre ellos, se escoge uno aleatorio, y, al igual que el otro método, lo buscará y reproducirá si lo encuentra.

Por último, para invocar a estos métodos, basta con usar la instancia estática del *singleton* de *ControladorSonido,* por ejemplo, con la siguiente línea:

```
AudioManager.instance.PlaySFX(”DoorOpen”);
```

Y para reproducir un sonido de entre varios, primero hay que definir un array de *string*:

```
string[] effects = { ”Zombie1”, ”Zombie2”, ”Zombie3” };
AudioManager.instance.PlayrandomSFX(effects);
```

Creación de audio para el juego

Ahora que sabemos cómo funciona el sistema de audio en *Into the Crypt*, veremos cómo se han creado y editado algunos de los sonidos del juego.

La mayoría de los audios no se han grabado, si no que se han usado audios con licencias que permitan o bien la edición o bien la distribución o ambos. En el caso de aquellos cuya licencia *Creative Commons* permite la edición se ha utilizado Audacity para dicho fin.

Edición de efectos de sonido

Para editar los efectos de sonido se han aplicado técnicas de recorte, cambios de envolvente, desvanecimientos y de reducción de ruido con el fin de adaptar los sonidos a los requisitos del juego.

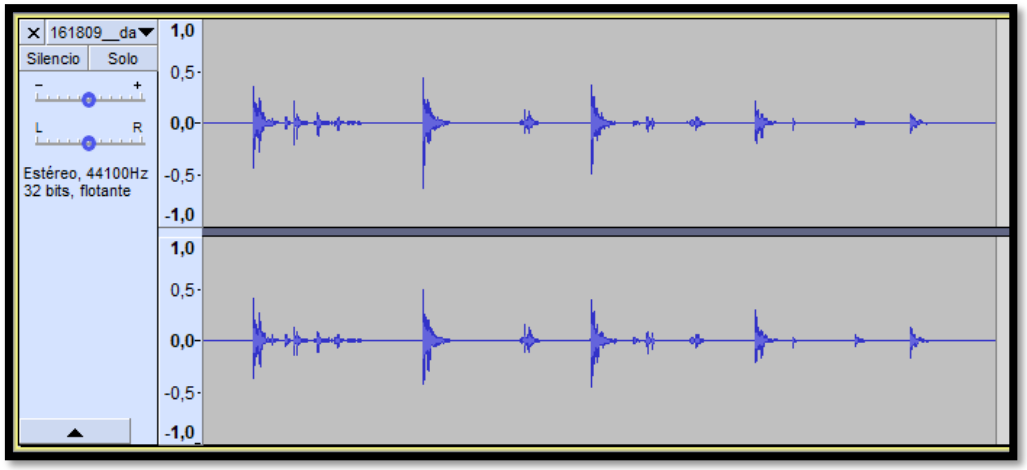


Figura 131 Sonido de una persona subiendo por una escalera

Fuente: elaboración propia

Podemos ver que la onda de este efecto de sonido tiene muchas inconsistencias en la forma de picos de diferentes alturas, ruido después de los principales golpes de sonido, etc.

Tras aplicar las siguientes técnicas:

- Recorte
- Reducción de ruido
- Normalización
- Cambio de tono
- Ecualización

Llegamos al siguiente sonido:

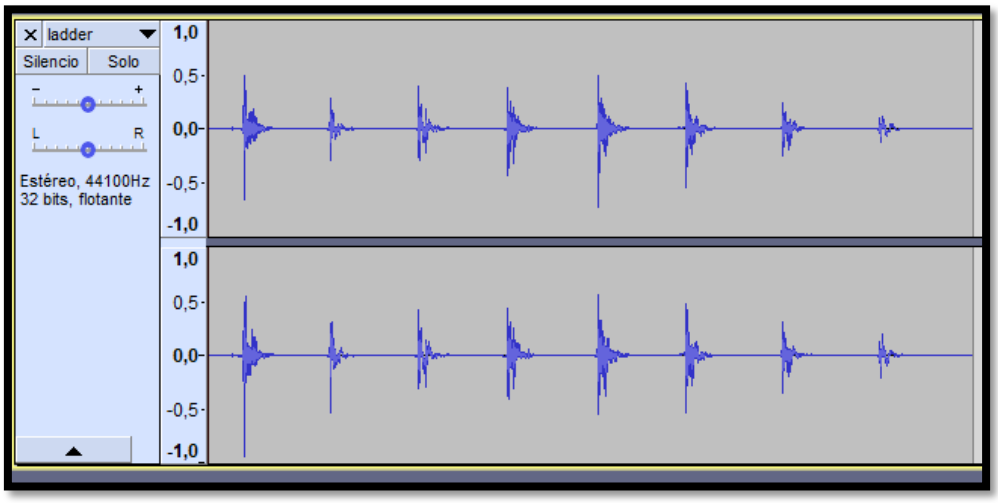


Figura 132 Sonido de una persona subiendo por una escalera editado

Fuente: elaboración propia

Este sonido, al tener los golpes de sonido más anchos y los picos de amplitud más altos, suena más grave, lo cual se acerca más al efecto que produciría escalar una escalera de madera vieja que pudiese estar en una cripta abandonada.

Creación de música

Para la creación de música se ha utilizado Bosca Ceoil, el cual permite crear desde *chiptunes* hasta composiciones orquestales de gran escala. Sin embargo, esta ha demostrado ser una tarea muy difícil, por lo que la música principal del juego se ha extraído de internet.

Aun así, algunos efectos musicales, como la canción que suena al subir de nivel, han sido compuestos usando Bosca Ceoil.

Figura 133 Composición de una canción en Bosca Ceoil

Fuente: elaboración propia

Implementación del tutorial

El nivel de tutorial tiene lugar en una escena aparte y sigue el guion definido en el GDD.

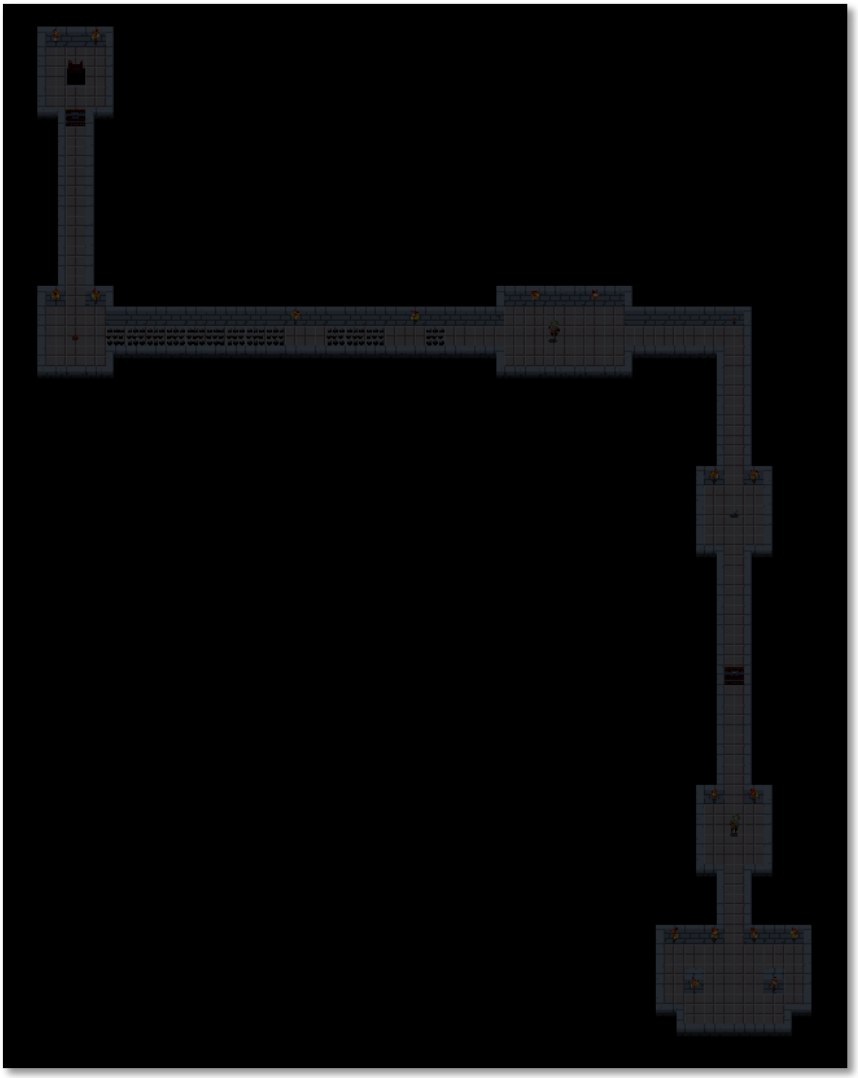


Figura 134 Mapa del tutorial

Fuente: elaboración propia

Además, se han añadido textos que flotan dentro del mundo para integrarlos con el nivel más, que indican al jugador que hacer y cómo.

Figura 135 Textos de ayuda en el tutorial

Fuente: elaboración propia

A continuación, se mostrará el recorrido del tutorial.

Figura 136 Primera habitación del tutorial

Fuente: elaboración propia

En esta primera habitación, el jugador aprenderá como moverse por el mapa. Un jugador que preste atención a su entorno notará que una de las antorchas está apagada. Al acercarse, el jugador verá aparecer un icono sobre la cabeza del personaje, indicándole un botón que pulsar para interactuar. Cabe destacar que este icono aparece con todos los objetos con los que el

jugador puede interactuar y que el icono cambia dependiendo de si el jugador está jugando con un mando o con el teclado solamente.

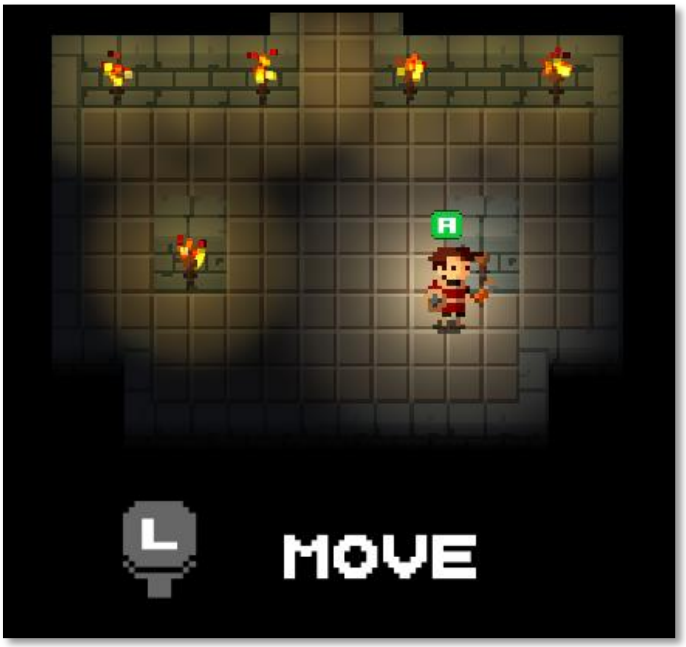


Figura 137 Ejemplo de interfaz adaptativa al periférico.

Fuente: elaboración propia

Figura 138 Segunda habitación del tutorial

Fuente: elaboración propia

En la segunda habitación el jugador aprenderá a pelear contra un enemigo que le perseguirá lentamente y que no atacará.

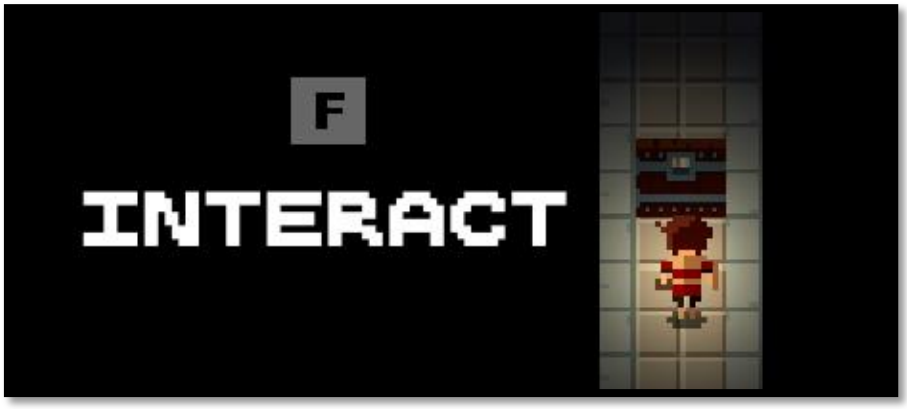


Figura 139 Tercer paso del tutorial

Fuente: elaboración propia

Más adelante, el jugador se encontrará con una puerta con la que podrá interactuar para abrirla y continuar. Si el jugador exploró y encendió la antorcha de la primera habitación, el concepto de interactuar le será familiar. En caso contrario, un texto le dirá como hacerlo.

Figura 140 Cuarta fase del tutorial

Fuente: elaboración propia

En esta habitación, el jugador pondrá a prueba lo aprendido hasta el momento y tendrá que interactuar con la espada del suelo para recogerla. Un texto le explicará cómo abrir el inventario, y, al hacerlo, recibirá una ayuda sobre como equipar o descartar un objeto. Esta ayuda aparece siempre.

Figura 141 Quinta fase del tutorial

Fuente: elaboración propia

Al seguir avanzando, el jugador encontrará un enemigo inmóvil que atacará constantemente. De esta forma, podrá probar su habilidad de bloqueo. El jugador puede, además, atacar a este enemigo para derrotarlo y conseguir algo de experiencia.

Figura 142 Sexto paso en el tutorial

Fuente: elaboración propia

Al llegar a este pasillo, el jugador se encontrará con una serie de trampas dispuestas de manera diferentes:

- Una trampa que podrá sortear corriendo.
- Tres trampas que podrá sortear solo si esprinta. El jugador, además, habrá observado el patrón de las trampas y sabrá cuando atravesar la zona.
- Nueve trampas insorteables, el jugador tendrá que recibir daño para continuar. Aunque el jugador espere para recargar su energía e intente esprintar dos veces para sortear esta serie de trampas, no llegará al final sin recibir daño.

Figura 143 Penúltima habitación del tutorial

Fuente: elaboración propia

La anterior fase asegura que el jugador llegue dañado a esta habitación. En ella el jugador recibirá una poción y, al beberla, recuperará la salud perdida. El jugador, sin importar cuantos golpes reciba del *zombie* o cuantas trampas pise, no podrá morir en el tutorial. Su salud solo podrá bajar hasta llegar a un punto de salud.

Figura 144 Final del tutorial

Fuente: elaboración propia

En esta última habitación, el jugador encontrará una escalera con la que podrá interactuar para continuar y terminar el tutorial. Además, se le enseñará como abrir el mapa y como pausar el juego, aunque, al abrir el mapa en el tutorial no podrá verlo y recibirá un mensaje indicándole que no hay mapa disponible de la zona.

Problemas encontrados y soluciones aplicadas

Ya que esta iteración estaba centrada en añadir contenido, no he encontrado demasiados problemas, a excepción de la composición de música, debido a lo complicada que es dicha tarea.

Sexta iteración

El objetivo de esta iteración es mejorar el *look and feel* del juego, equilibrarlo, optimizarlo y arreglar todos los errores posibles. Cuando estas tareas estén terminadas, se publicará el juego y se dará por terminada tanto la iteración como la etapa de desarrollo.

Optimizaciones

De nada sirve que un juego sea divertido, tenga mecánicas profundas o tenga un aspecto visual maravilloso si el rendimiento del juego es deficiente. Para solucionar tal problema, los juegos son sometidos a diversas etapas de optimizaciones a lo largo de la etapa de desarrollo.

Aunque durante todo el desarrollo ha habido múltiples optimizaciones menores tanto en el código como en el almacenamiento de recursos como *sprites* o sonidos, nos centraremos en las optimizaciones de mayor importancia que ha sufrido *Into the Crypt.*

Optimización del uso del gestor de recursos

Al principio, y, al igual que en la actualidad, las llamadas al gestor de recursos se realizaban en el método *LateUpdate* del componente *AspectoAnimación*, tal y como se vio en la sección *El gestor de contenidos de Unity*. El problema que tenía en esta primera iteración era que la carga de *sprites* se hacía también en *LateUpdate* por lo que cada *frame* se estaba cargando y descargando los mismos *sprites* en memoria, aunque no hiciese falta. Esto, además, provocaba fragmentación del disco y que en repetidas ocasiones apareciese el recolector de basura o *Garbage Collector* disponible en C#.

Aunque el recolector de basura se encarga de gestionar la memoria en lugar del desarrollador, si se cometen errores de gestión demasiado grandes, el recolector se verá obligado a pasar varias veces y con mucha frecuencia, y, ya que las operaciones que realiza son lentas, el rendimiento del juego se ve afectado y provoca parones en el *framerate.*

Para solucionar este problema se introdujo el código mostrado anteriormente, en el que el componente *AspectoAnimación* almacena los *sprites* actuales y solo carga mediante el gestor de recursos una nueva colección de *sprites* cuando es necesario, reduciendo tanto el tiempo de ejecución del componente como los problemas de memoria.

Gracias al *profiler* que el editor de Unity nos proporciona podemos monitorizar el rendimiento del juego y hacer comparativas entre diferentes etapas del desarrollo. Por ejemplo, antes y después de implementar este cambio.

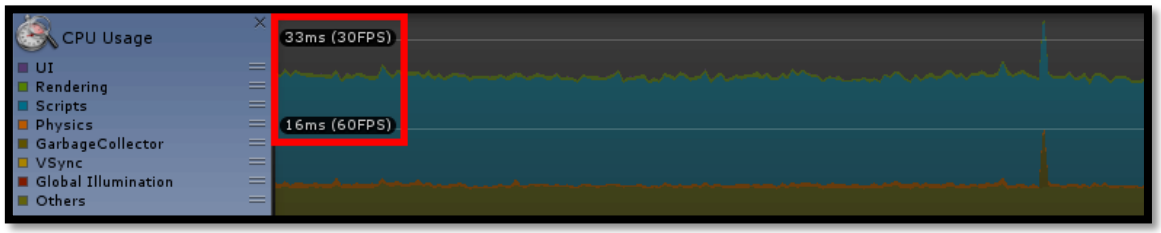


Figura 145 Rendimiento del juego antes de optimizar las llamadas al gestor de recursos

Fuente: elaboración propia

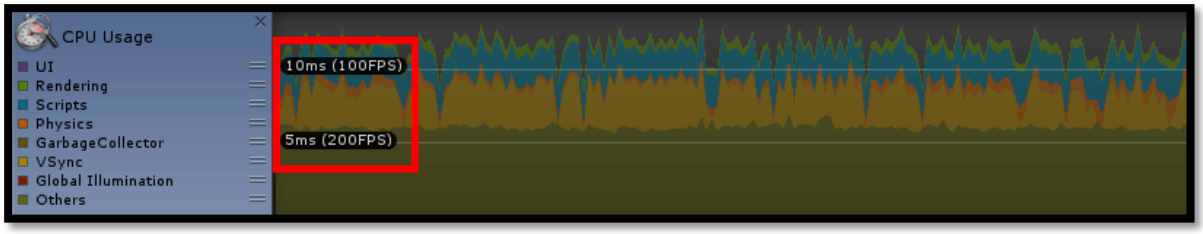


Figura 146 Rendimiento del juego tras optimizar las llamadas al gestor de recursos

Fuente: elaboración propia

Color	**Tipo de proceso**
	Interfaz
	Renderizado
	Lógica del juego
	Calculo de físicas
	Recolector de basura
	Sincronización vertical
	Iluminación global
	Otros

Tabla 15 Leyenda del profiler de Unity

Antes de empezar, destacaremos que los procesos de tipo Otros normalmente no tienen nada que ver con el juego, sino con el editor de Unity. Por ejemplo, en estas figuras vemos alrededor de 5 ms siendo consumidos por proceso de tipo Otros. Se debe principalmente al propio profiere recolectando información

En cuanto al resto de procesos, vemos como en la figura 145 el juego se ejecuta alrededor de cuarenta veces por segundo. Podemos ver cerca del margen derecho de la figura un pico en el juego baja a menos de 30 *frames* por segundo debido a la ralentización que causa el recolector de basura.

Sin embargo, al optimizar el código vemos el tiempo de ejecución de la lógica del juego se reduce enormemente, haciendo que el *framerate* oscile alrededor de las 100 *frames* por

segundo. Vemos también como el juego usa ahora sincronización vertical. Esto ocurre cuando todo el código del juego, incluyendo cálculos de físicas, lógica del juego, gestión de memoria, etc.; termina antes de que la pantalla haya terminado de renderizar, por lo que rellena el tiempo mientras espera con sincronización vertical.

Optimización del mundo de juego: *culling*

En un juego en el que el mundo es de un tamaño considerable, el uso de *level of detail* o nivel de detalle es una técnica comúnmente aplicada. Generalmente la podemos dividir en dos categorías:

- *Level of detail* para gráficos: los objetos lejanos se verán con menos detalle que aquellos cercanos.
- *Level of detail* para lógica del juego: aquellos objetos lejanos ejecutaran su lógica menos frecuentemente.

Si vamos un paso más allá, llegaremos a la técnica del *culling*. Aquellos objetos que no se pueden ver no serán dibujados y no ejecutarán su lógica. Aunque esto parece lógico, no podemos realmente saber si los objetos fuera de la cámara hacen esto realmente, ya que el usuario no puede ver las cosas que están fuera de la pantalla, y, aunque los motores gráficos no renderizan aquellos objetos fuera de cámara, sí que reciben la información de dichos objetos, con la que se calcula si están a la vista y si se deberían renderizar. En cuanto a la lógica, ningún sistema se encarga de desactivar su ejecución de forma automática, sino que es tarea del desarrollador implementarlo.

Por eso, para optimizar el rendimiento, se ha incluido en *Into the Crypt* un sistema de *culling* para desactivar todos aquellos objetos a más de una habitación de distancia. La comprobación y la fase de activación y desactivación se lleva a cabo cada vez que el jugador cambia de habitación.

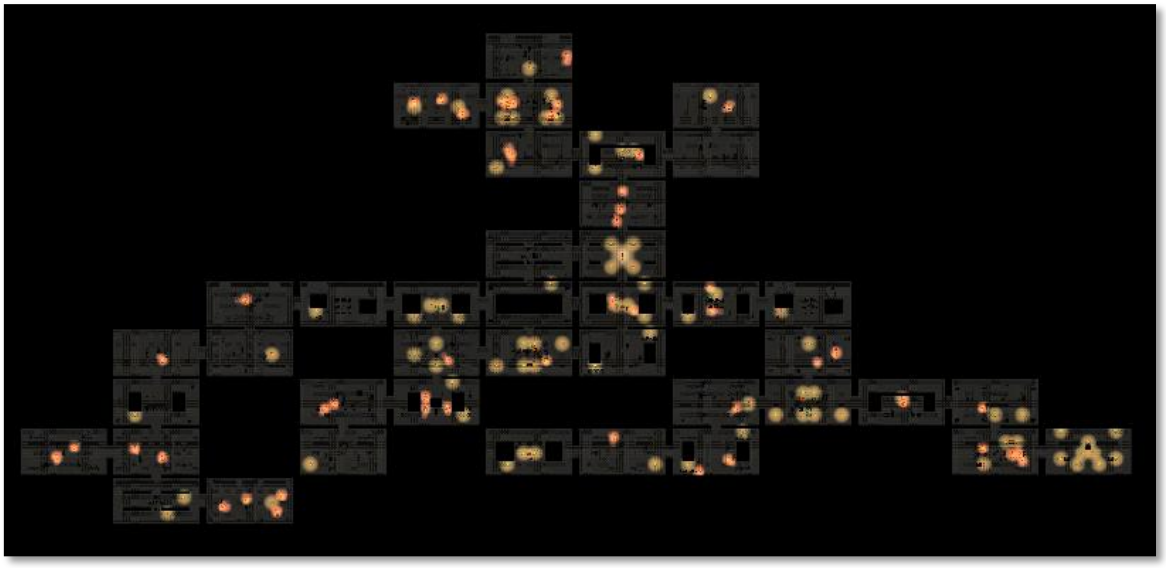


Figura 147 El mundo real del juego

Fuente: elaboración propia

Figura 148 El mundo del juego, tras aplicar la técnica del culling

Fuente: elaboración propia

Gracias a las dos figuras anteriores podemos apreciar el número de objetos que el juego ya no tiene que renderizar y la lógica y luces que ya no tiene que calcular. Aun así, vamos a comparar el rendimiento del juego con el *profiler*, aunque en el estado real del juego, ya que para las anteriores figuras se ha modificado el tamaño de la cámara por el bien de la explicación.

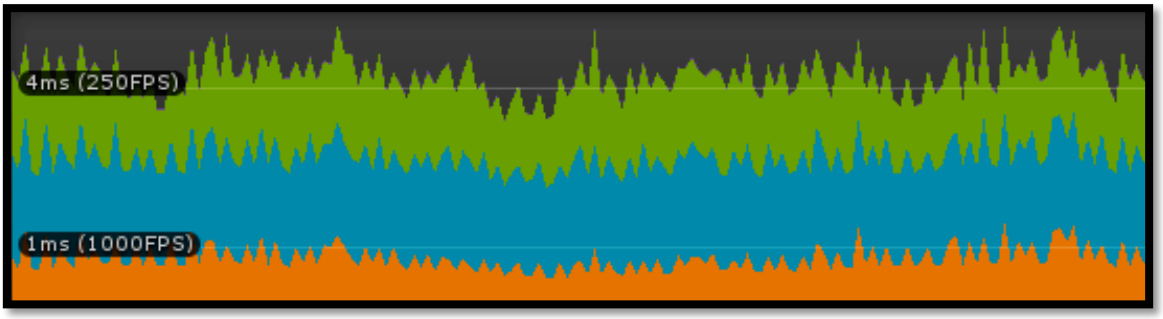


Figura 149 Rendimiento del juego antes aplicar técnicas de culling

Fuente: elaboración propia

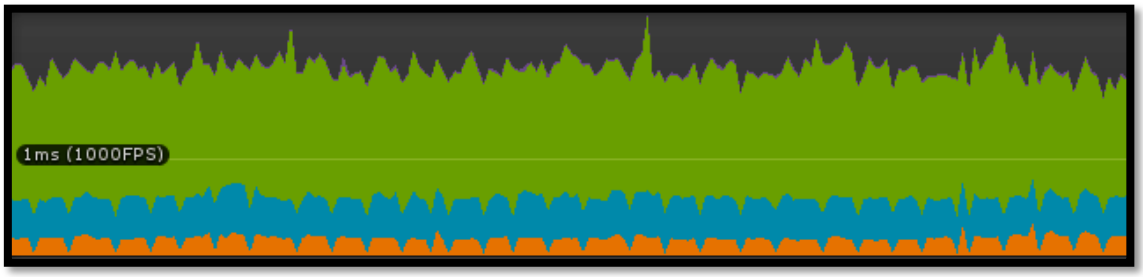


Figura 150 Rendimiento del juego después de aplicar técnicas de culling

Fuente: elaboración propia

En esta ocasión se han obviado tanto los procesos del tipo Otros como la sincronización vertical. Podemos apreciar que al aplicar *culling*, el tiempo tanto de renderización, como de cálculo de físicas y lógica se reduce de manera drástica. Con esta optimización se ha reducido el tiempo de ejecución de cada ciclo en alrededor de 3 ms.

Finalmente, la razón por la cual no se aplica *culling* a las habitaciones colindantes a la actual es para que el mundo siga en movimiento y cambiante. Ya que se espera que el jugador cambie entre esas habitaciones más frecuentemente, si se encontrará siempre todo tal y como estaba la última vez que lo vio, no daría esa sensación.

Transiciones de cámara

En los RPG se suelen utilizar dos tipos de cámara diferentes:

- Cámaras de seguimiento que, como su nombre indica, siguen al jugador captando el mundo a su alrededor.
- Cámaras que captan el mundo desde un punto fijo.

En *Into the Crypt* conviven los dos tipos de cámara: el primero en el tutorial y el segundo en el resto del juego. En esta sección nos centraremos en el segundo tipo.

En las mazmorras de *Into the Crypt*, la cámara se mantiene fija, enfocando la habitación en la que el jugador se encuentra. Así, al cambiar el jugador de habitación, la cámara debería hacer lo mismo, pero, simplemente cambiar la posición de la cámara puede provocar mareo y desorientación al jugador. Por eso, se utilizan transiciones que desplacen la cámara de un punto a otro durante un intervalo de tiempo. Con este fin se ha desarrollado un componente al que hemos llamado *CamaraHabitacion.*

CamaraHabitacion se define por dos atributos:

- *DuracionTransicion* define cuanto tiempo tardará en desplazarse la cámara de una posición a otra.
- *CurvaPosicion* define como la cámara realizara el movimiento a lo largo del tiempo. Esto se hace con una curva de animación y permite modificar el movimiento en función de la curva que elijamos.

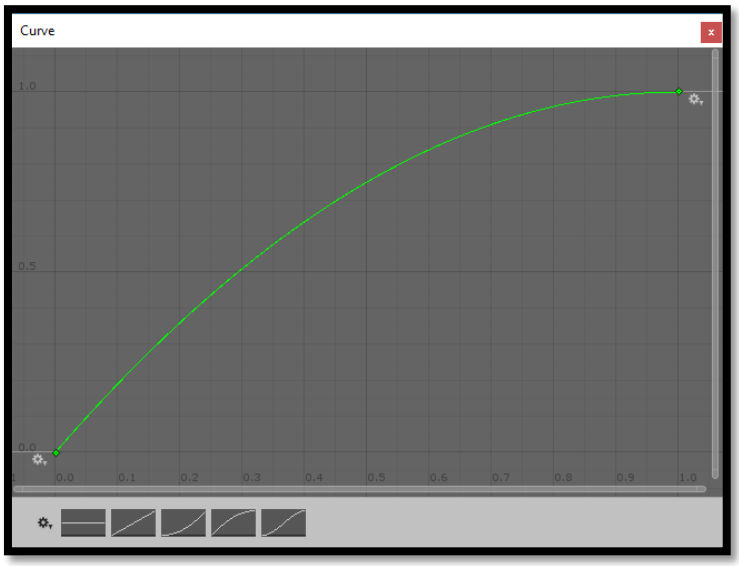


Figura 151 Curva de animación en el editor de Unity

Fuente: elaboración propia

El funcionamiento de este componente se basa en el siguiente método:

```
    IEnumerator MoveCamera(Vector3 endPos, float wait) {
        if (wait != 0)
            yield return new WaitForSeconds(wait);

        Vector3 StartPos;
        StartPos = transform.localPosition;
        float time = 0;
        float percent = 0;
        float lastTime = Time.realtimeSinceStartup;
        do {
            time += Time.deltaTime;
            percent = Mathf.Clamp01(time / transitionTime);
            Vector3 tempPos = Vector3.LerpUnclamped(StartPos,
endPos, posCurve.Evaluate(percent));
            transform.localPosition = tempPos;
            yield return null;
        } while (percent < 1);
        // Make sure it ends on the desired position.
        transform.localPosition = endPos;
        transitionTime = actualDuration;
        yield return null;
    }
```

Este método funciona de una manera un tanto peculiar debido a que es una corrutina. Ejecutar una corrutina es similar a lanzar un hilo en un procesador multihilo para ejecutar una función. Sin embargo, el funcionamiento es diferente ya que al ejecutar la instrucción:

```
yield return null; //No necesariamente null
```

Unity guarda una referencia a la posición del código en la que se ejecutó esta línea y abandona la ejecución durante el resto del ciclo actual. Al llegar al siguiente ciclo, Unity retomará la ejecución desde donde la referencia almacenada le indique.

En cuanto al método, su funcionamiento es básicamente analizar cuanto tiempo ha pasado, y, mientras este tiempo sea menor a la duración definida, analizar la curva y aplicar el cambio de posición. De esta forma conseguimos de una manera sencilla transiciones fluidas que mejoran tanto la usabilidad del juego como el *look and feel*.

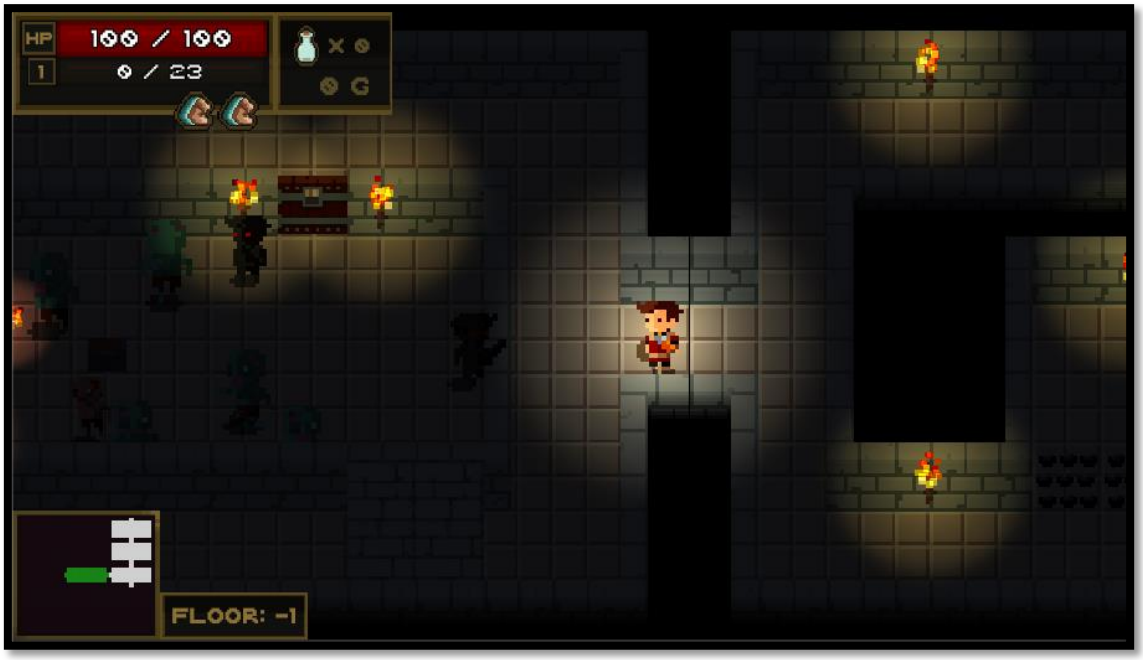


Figura 152 Una transición entre habitaciones a medias

Fuente: elaboración propia

Animación del menú principal

Para hacer el menú principal más animado, se ha creado una animación en la que, en el fondo, detrás del menú, se ven algunas habitaciones con enemigos y objetos que se van desplazando de derecha a izquierda constantemente. En realidad, solo la cámara se mueve de una habitación a otra y, al llegar a la última habitación, las otras seis se colocarán al otro lado para producir el efecto de infinidad.

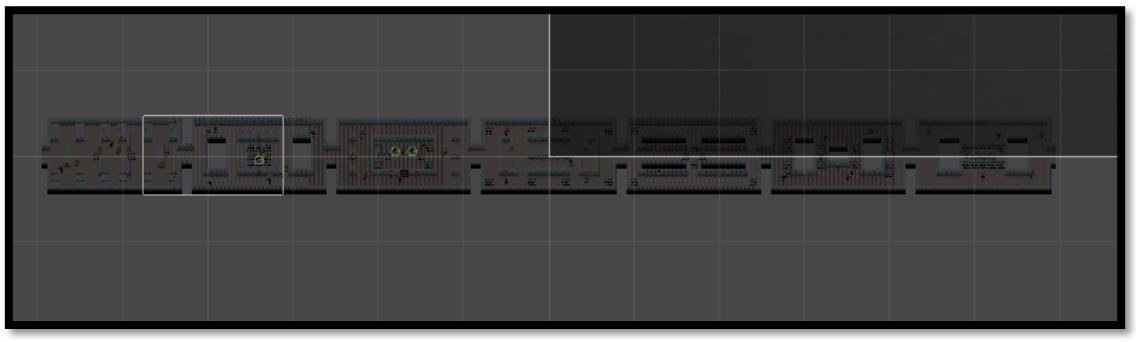


Figura 153 La cámara pasando de la primera a la segunda habitación

Fuente: elaboración propia

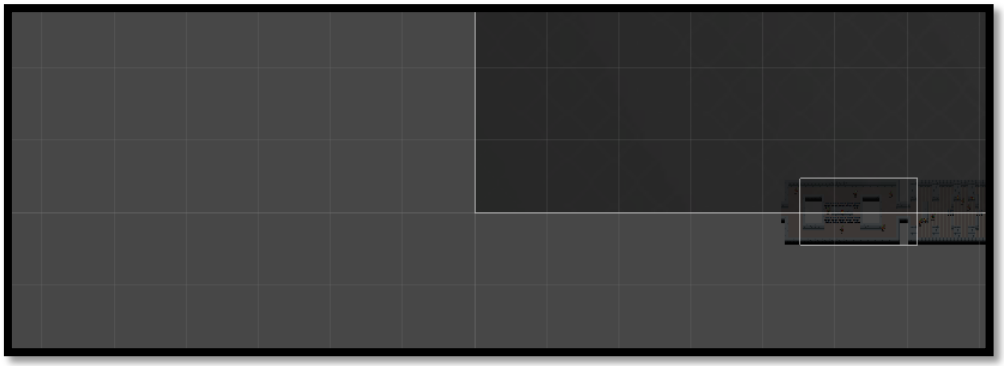


Figura 154 La cámara pasando de la habitación siete a la habitación número uno

Fuente: elaboración propia

Esta animación reutiliza el controlador de la cámara del juego.

Partículas

En un juego, las partículas aumentan mucho el sentimiento de que el mundo está vivo y activo, ya que se comportan aleatoriamente por naturaleza. Además, abaratan costes ya que todo aquello que se pueda animar mediante partículas es trabajo que el encargado del arte no tiene que hacer. Las antorchas de *Into the Crypt* son un ejemplo de esto:

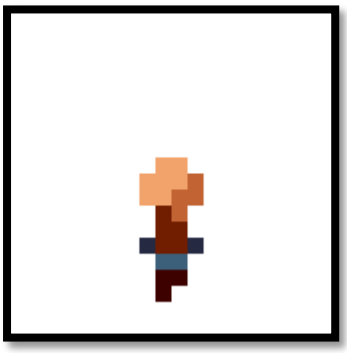


Figura 155 Sprite de una antorcha

Fuente: elaboración propia

Figura 156 Una antorcha encendida

Fuente: elaboración propia

Sin embargo, las partículas no solo mejoran el trabajo de desarrollo, sino que también pueden ayudar a mejorar la claridad del juego. En *Into the Crypt* esto se hace a través de partículas de sangre al ser golpeado.

Figura 157 El protagonista sangra al ser golpeado

Fuente: elaboración propia

Vemos que, a pesar de ser un efecto muy sutil, da mucha claridad y vida al juego.

Detalles en los *sprites*

Aunque en la sección anterior hemos hablado sobre como las partículas ayudan a mejorar el *look and feel* de un juego sin aumentar el trabajo que se dedica al apartado de arte de este, hay mucho que se puede hacer en cuanto a animación y diseño de *sprites* para fortalecer el apartado gráfico.

En el mundo del PixelArt es común hacer animaciones en las que los personajes reboten mucho, ya que los rebotes contrastan con la normalmente baja resolución de los *sprites* en PixelArt y proporcionan una sensación de vitalidad a la animación.

Aunque añadir sensación de vitalidad es difícil en unos *sprites* con tan pocos píxeles como los de *Into the Crypt*, se puede conseguir mediante el uso de *keyframes.*

Las *keyframes* son *frames* específicas en una animación y son las encargadas principales de definir el movimiento. Todas las *frames* que no son *keyframes* se denominan *in-between frames*.

Figura 158 Ejemplo de keyframes en una animación de ataque

Fuente: elaboración propia

Esta animación tiene tres *frames*. Vamos a analizarlas una a una:

1. El personaje se prepara para atacar. Echa el cuerpo para atrás, repliega el escudo y prepara el arma.
2. Esta es una *in-between frame*. La espada se ha movido tan rápido que ha dejado un rastro por donde ha pasado. Además, brilla con un blanco resplandeciente. El pelo del personaje también reacciona antes la brusquedad del movimiento.
3. El ataque ha terminado. El rastro del arma se desvanece y el brillo se atenúa hasta la normalidad.

Así, podemos ver como con tres *frames* solamente se ha transmitido tanta sensación de movimiento.

También es posible que una animación no tenga *in-between frames*, es decir, que se componga solo de *keyframes.* Por ejemplo, la siguiente animación.

Figura 159 Ejemplo de una animación compuesta solamente por keyframes

Fuente: elaboración propia

En esta animación podemos ver que todas las *frames* representan un momento del movimiento. Sin embargo, aunque sea una sencilla animación de caminar, podemos ver como el ojo que cuelga de la cuenca del *zombie* se balancea de un lado a otro. Apreciamos también como el brazo izquierdo del *zombie* parece colgar, dando el efecto de dislocación y dando la impresión de que este *zombie* tiene ese brazo roto.

Teoría del color

En el anterior apartado hemos visto que combinando el detalle en los *sprites* con un buen aprovechamiento de un sistema de partículas podemos conseguir que nuestro juego tenga un buen aspecto gráfico. Sin embargo, todo esto es en vano si los colores escogidos para nuestros gráficos son incorrectos.

Aquí entra en juego la teoría del color. La teoría del color es un conjunto de reglas en la mezcla de colores que determinan cómo conseguir el nivel perfecto de luminosidad y pigmentación [47]. La regla principal de la teoría del color cuando esta se aplica al PixelArt es que para conseguir luces y sombras no hay que cambiar la luminosidad si no el tono o pigmento.

Figura 160 Árbol en Secret o f Mana

Fuente: gas13.ru

En este ejemplo de un árbol, vemos que las zonas sombrías del árbol (centrándonos solo en la parte verde) son de colores menos saturados mientras que los destellos son de un color amarillo vibrante. Esto da al mundo del juego una sensación de vida. En *Into the Crypt* se ha diseñado una paleta en la que, aunque los colores cambian de tono dependiendo de si representan una parte clara o una oscura, el cambio es más sutil que en el ejemplo anterior.

Figura 161 Paleta de colores usados en Into the Crypt, ordenados por materiales

Fuente: elaboración propia

Esta paleta de colores de elaboración propia tiene, en su gran mayoría, colores con poca saturación. Están, además, ordenados por el material que representan (carne, tela, metal, etc.)

Para probar que los colores de una paleta han sido escogidos correctamente, hay que pasar una imagen a escala de grises y comprobar si se diferencian claramente los objetos.

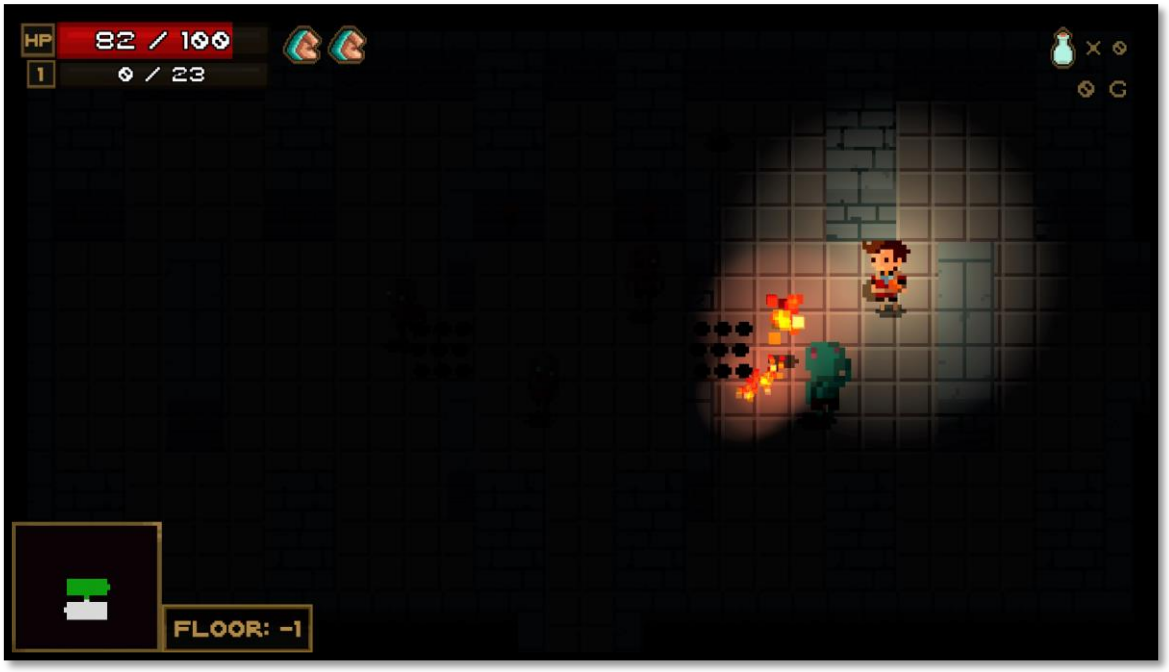


Figura 162 Imagen a color de Into the Crypt

Fuente: elaboración propia

Figura 163 Imagen en escala de grises de Into the Crypt

Fuente: elaboración propia

En la imagen en escala de grises vemos que todos los personajes resaltan en comparación con el escenario. Se puede apreciar también que los personajes tienen zonas sombreadas y otras más iluminadas y sus atuendos están compuestos de materiales de diferentes colores.

Podemos determinar entonces que la teoría del color está bien aplicada y los colores escogidos son correctos.

Movimiento de los objetos

Al principio, los objetos que caían al suelo, por ejemplo, al abrir un cofre, aparecían simplemente en el lugar determinado. Para darle más realismo y vida a esto se ha implementado un componente que, mediante curvas de Bézier cubicas se encarga de mover los objetos que caen. Además, ya que la principal intención es simular un movimiento 3D, se ha hecho que con curvas de Bézier lineales se mueva la sombra a lo largo de lo que sería el plano del suelo.

Para que un objeto parezca volar por el escenario, vamos a aplicar la fórmula de las curvas cubicas de Bézier.

$$B(t) = P_0(1-t)^3 + 3P_1t(1-t)^2 + 3P_2t^2(1-t) + P_3t^3,\ t \in [0,1]$$

En esta fórmula, P_0 y P_3 son el punto de inicio y final (elegido aleatoriamente), respectivamente. P_1 y P_2 son los nodos de anclaje que definen la curva. Estos puntos son iguales a P_0 y P_3 en todas sus componentes excepto en la Y, en la que se le suman dos unidades.

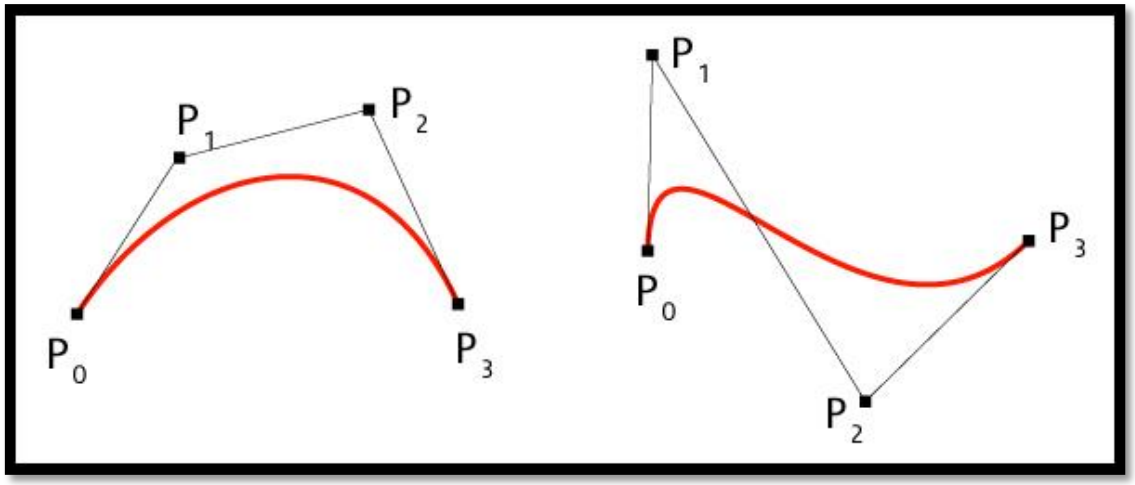


Figura 164 Ejemplos de curvas de Bézier cúbicas

Fuente: www.e-cartouche.ch

Para mover la sombra del objeto por el suelo se aplicará la fórmula de las curvas de Bézier lineales.

$$B(t) = P_0 + (P_1 - P_0)t$$

De esta forma, obtendremos un resultado como el siguiente:

Figura 165 Objetos cayendo al suelo

Fuente: elaboración propia

Podemos apreciar que da la sensación de que el hacha y las monedas que acaban de salir del cofre están flotando en el aire mientras caen al suelo.

Problemas encontrados y soluciones aplicadas

Durante esta iteración no he encontrado problemas, ya que, al trabajar sobre todo en funcionalidades ya implementadas y que solo había que retocar un poco, no ha supuesto problema alguno en cuanto a implementación y, al trabajar en partes del juego generalmente desacopladas, tampoco ha habido problemas de integración.

Promoción y publicación del juego

Una vez que el juego está terminado es el momento de promocionarlo y publicarlo.

Promoción en redes sociales

A mediados de la fase de desarrollo se creó una cuenta de Twitter para promocionar el juego. Con ella se mostró al mundo gifs e imágenes del progreso. Aquellos que seguían la cuenta, daban *retweet* y "Me gusta", lo que ayudaba a promover aún más el videojuego.

El día de la publicación, la cuenta contaba con un total de 184 seguidores y con 67 *tweets* se consiguieron las siguientes estadísticas:

- 86 *retweets*.
- 400 "Me gusta"
- 41600 impresiones, es decir, gente en cuyo *feed* de Twitter ha aparecido un tweet de esta cuenta.

Aunque no son números demasiado grandes (a excepción de las impresiones, que es una medida irrelevante), han sido suficientes para motivarme a continuar desarrollando.

Publicación del juego

Antes de publicar el juego, se ha creado un tráiler usando Adobe Premiere y se ha subido a YouTube [48].

Figura 166 Edición del tráiler en Adobe Premiere

Fuente: elaboración propia

Una vez listo el tráiler del juego, se hicieron varias capturas del juego y, tras registrarse en itch.io se configuró una página para *Into the Crypt* con una pequeña introducción, las imágenes y el tráiler y un enlace de descarga del juego [49].

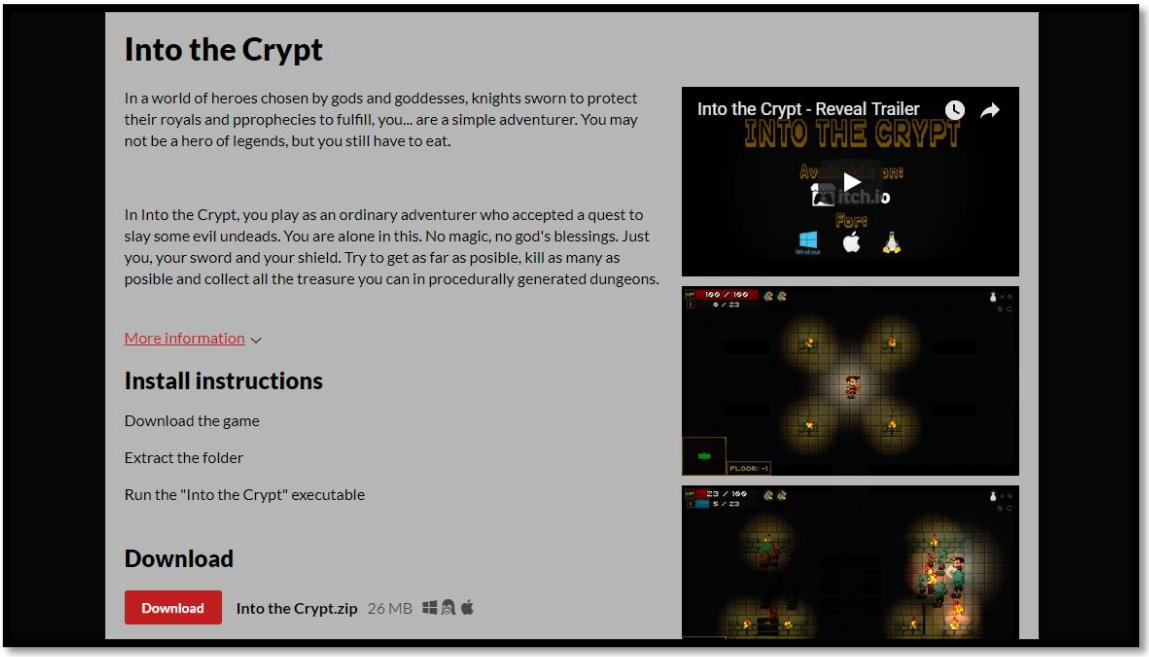


Figura 167 Pagina de Into the Crypt en itch.io

Fuente: elaboración propia

Tras la publicación del juego, se anunció tanto en la cuenta de Twitter oficial del juego como a través de mi cuenta personal. Además, envié los enlaces a diversos conocidos a través de aplicaciones de mensajería.

Conclusiones

A lo largo de este año he aprendido muchas cosas relativas al mundo de los videojuegos y su desarrollo. Tanto en las clases de la universidad, desarrollando el ABP, como en las prácticas de empresa en las que pasé tres meses trabajando con Unity. Todo esto, combinado con mi propia experiencia previa con Unity (adquirida de forma autodidacta) y programando han dado lugar a un producto del cual me siento orgulloso. No solo como para exponerlo ante el tribunal, sino también como para publicarlo y enseñárselo al mundo.

Al principio de la etapa de desarrollo, quería implementar el juego de mis sueños: un RPG que rindiese honor a clásicos del género, pero que adoptara también las mecánicas modernas que hacen que hoy en día los juegos RPG sigan triunfando en el mercado. Sin embargo, no solo he conseguido eso, sino que he ido más allá. Jamás habría imaginado que el juego tendría un resultado tan bueno como el que tiene.

En cierto modo, se puede notar cierto paralelismo entre este proyecto y la carrera de ingeniería multimedia. Al principio esperas llegar a la cima y esforzarte para conseguir tus sueños, pero, conforme avanzas, te das cuenta de que no es un camino fácil. Pese a esto, decides seguir adelante, con esfuerzo y pasión, y, una vez llegas al final, te encuentras con unas fantásticas habilidades bajo tu brazo, habilidades que jamás esperaste conseguir. Y, al parar un momento, te das cuenta de que la cima que buscabas, hace tiempo que la superaste.

Esta carrera, todos los alumnos con los que he convivido y los profesores que he conocido me han ayudado a ser el ingeniero que soy hoy. Y creo que este proyecto, que tan bien representa la experiencia de cursar ingeniería multimedia es un hermoso broche final que poner a mi paso por la universidad de Alicante mientras estudiaba este grado.

Con este proyecto, además, he mejorado mis capacidades tanto de programación como de diseño. He conseguido aprovechar todo el potencial que Unity ofrece y he conseguido un producto del cual sentirme orgulloso y que puedo utilizar como carta de presentación en el mundo laboral. *Into the Crypt* marca el inicio de mi etapa en este mundo y estoy seguro de que los conocimientos que he conseguido gracias a él me abrirán muchas puertas, aunque aún hay mucho que mejorar y seguir aprendiendo.

Bibliografía y referencias

Material consultado

1. *US video game industry revenue reaches $36 billion in 2017*. Disponible en www.theesa.com: **http://www.theesa.com/article/us-video-game-industry-revenue-reaches-36-billion-2017/**
2. *Mobile revenues account for more tan 50% of the global games market as it reaches $137.9 billion in 2018*. Disponible en www.newzoo.com: **https://newzoo.com/insights/articles/global-games-market-reaches-137-9-billion-in-2018-mobile-games-take-half/**
3. *Role-playing video game*. Disponible en www.wikipedia.org: **https://en.wikipedia.org/wiki/Role-playing_video_game**
4. *The future of the ARPG*. Disponible en www.gamasutra.com: **https://www.gamasutra.com/blogs/StiegHedlund/20160505/272009/The_Future_of_the_ARPG.php**
5. *Why are modern roguelike games so popular?* Disponible en www.gamedevlibrary.com: **https://gamedevlibrary.com/gamedev-thoughts-why-are-modern-roguelike-games-so-popular-53c1f5a05485**
6. *Roguelikes: The rebirth of the counterculture*. Disponible en www.ign.com: **http://www.ign.com/articles/2014/07/04/roguelikes-the-rebirth-of-the-counterculture**
7. *CryEngine*. Disponible en www.wikipedia.org: **https://es.wikipedia.org/wiki/CryEngine**
8. *CryEngine V Manual*. Disponible en docs.cryengine.com: **http://docs.cryengine.com/display/CEMANUAL/CRYENGINE+V+Manual**
9. *CryEngine Showcase*. Disponible en www.cryengine.com: **https://www.cryengine.com/showcase**
10. *Xenko*. Disponible en www.wikipedia.org: **https://en.wikipedia.org/wiki/Xenko**
11. *Xenko Showcase*. Disponible en fórums.xenko.com: **https://forums.xenko.com/c/showcase**
12. *Unreal Engine Awards*. Disponible en www.unrealengine.com: **https://www.unrealengine.com/en-US/awards**

13. *Games using Unreal Engine 4*. Disponible en www.giantbomb.com: **https://www.giantbomb.com/profile/wehrmacht/lists/games-using-unreal-engine-4/85929/**
14. *Godot Showcase*. Disponible en godotengine.org: **https://godotengine.org/showcase/1**
15. *Falcao Pearl Abs liquid V3*. Un plugin para el desarrollo de ARPG en RPG Maker con demostraciones en video. Disponible en falcaorgss.wordpress.com: **https://falcaorgss.wordpress.com/2012/12/02/falcao-pearl-abs-liquid-v1/**
16. *To the Moon*. Página de Steam de este juego que cuenta con un gran número de valoraciones extremadamente positivas. Disponible en store.steampowered.com: **https://store.steampowered.com/app/206440/To_the_Moon/**
17. *GameMaker Studio | 3D Dungeon Game, Dev Log 1*. Video mostrando un juego en 3D en GameMaker Studio, por Noone. Disponible en www.youtube.com: **https://www.youtube.com/watch?v=Xmw5LZuqzWQ**
18. *GameMaker Studio 2 Trial – Limitations.* Disponible en help.yoyogames.com: **https://help.yoyogames.com/hc/en-us/articles/230407528-GameMaker-Studio-2-Trial-Limitations**
19. *Game Maker Showcase.* Disponible en www.yoyogames.com: **https://www.yoyogames.com/showcase**
20. *This engine is dominating the gaming industry right now.* Disponible en thenextweb.com: **https://thenextweb.com/gaming/2016/03/24/engine-dominating-gaming-industry-right-now/**
21. *Netflix, Telltale* and "Stranger Things": Why the deal took two years to happen *(EXCLUSIVE)*. Disponible en variety.com: **https://variety.com/2018/gaming/features/netflix-telltale-stranger-things-deal-1202855424/**
22. *Made with Unity*. Disponible en unity.com: **https://unity.com/madewith?_ga=2.230799517.309102341.1532003021-1171346397.1517310322**
23. *Comunidad de CryEngine en Reddit.* Disponible en www.reddit.com: **https://www.reddit.com/r/cryengine/**
24. *Comunidad de Unreal Engine en Reddit.* Disponible en www.reddit.com: **https://www.reddit.com/r/unrealengine/**

25. *Comunidad de Godot en Reddit.* Disponible en www.reddit.com: **https://www.reddit.com/r/godot/**
26. *Comunidad de RPG Maker en Reddit.* Disponible en www.reddit.com: **https://www.reddit.com/r/RPGMaker/**
27. *Comunidad de GameMaker en Reddit.* Disponible en www.reddit.com: **https://www.reddit.com/r/gamemaker/**
28. *Comunidad de Unity3D en Reddit.* Disponible en www.reddit.com: **https://www.reddit.com/r/Unity3D/**
29. *Comunidad de Unity2D en Reddit.* Disponible en www.reddit.com: **https://www.reddit.com/r/Unity2D/**
30. *The 50 best video games of all time, according to critics.* Disponible en www.bussinessinsider.es: **https://www.businessinsider.es/best-video-games-metacritic-2017-11?page=50**
31. *List of videogames considered the best.* Disponible en en.wikipedia.org: **https://en.wikipedia.org/wiki/List_of_video_games_considered_the_best**
32. *Los 100 mejores juegos de la historia según EDGE.* Disponible en www.hobbyconsolas.com**: https://www.hobbyconsolas.com/noticias/100-mejores-juegos-historia-2017-segun-edge-162432**
33. *Top 25 Game Boy Advance games of all time.* Disponible en www.ign.com: **http://www.ign.com/articles/top-25-game-boy-advance-games-of-all-time**
34. *List of best-selling Game Boy Advance video games*. Disponible en en.wikipedia.org: **https://en.wikipedia.org/wiki/List_of_best-selling_Game_Boy_Advance_video_games**
35. DALE GREEN. *Procedural Content Generation for C++ Game Development.*
36. Dark Souls Challenge: Asylum Demon Broken Sword. Disponible en www.youtube.com: **https://www.youtube.com/watch?v=K2NnqEb0Xr0**
37. *Top 4 software development methodologies*. Disponible en www.synopsys.com: **https://www.synopsys.com/blogs/software-security/top-4-software-development-methodologies/**
38. *Software development process.* Disponible en www.wikipedia.org: **https://en.wikipedia.org/wiki/Software_development_process**
39. *Top 5 Software Development Methodologies.* Disponible en project-management.com: **https://project-management.com/top-5-software-development-methodologies/**

40. *Top 12 Software Development Methodologies & its Advantages / Disadvantages.* Disponible en www.tatvasoft.com: **https://www.tatvasoft.com/blog/top-12-software-development-methodologies-and-its-advantages-disadvantages/**
41. *All You Need To Know About Software Development Methodologies.* Disponible en usersnap.com: **https://usersnap.com/blog/software-development-methodologies/**
42. *Rapid ApplicationDevelopment (RAD): A Smart, Quick And Valuable Process For Software Developers*: Disponible en www.forbes.com: **https://www.forbes.com/sites/forbestechcouncil/2016/08/24/rapid-application-development-rad-a-smart-quick-and-valuable-process-for-software-developers/#63cd4af619e8**
43. *Metodología FDD (Feature Driven Development/ Desarrollo Basado en Funciones).* Disponible en metodologiafdd.com: **http://metodologiafdd.blogspot.com/**
44. *FDD; Its Processes & Comparison To Other Agile Methodologies*. Disponible en apiumhub.com: **https://apiumhub.com/tech-blog-barcelona/feature-driven-development/**
45. Caracteristicas y opiniones de HacknPlan. Disponible en hacknplan.com: **https://hacknplan.com/**
46. *From Usability to Playability: Introduction to Player-Centred Video game Development Process.* Disponible en link.springer.com: **https://link.springer.com/chapter/10.1007%2F978-3-642-02806-9_9**
47. *Teoría del color.* Disponible en www.wikipedia.org: **https://es.wikipedia.org/wiki/Teor%C3%ADa_del_color**
48. *Trailer de Into the Crypt.* Disponible en www.youtube.com: **https://www.youtube.com/watch?v=ZuNU20SIFeU**
49. *Página de Into the Crypt en itch.*io. Disponible en rafaelnavarro.itch.io: **https://rafaelnavarro.itch.io/into-the-crypt**

Recursos empleados

Para la realización de la memoria

1. *Spritesheet* de "*Isometric hero*" por Clint Bellager. Disponible en www.opengameart.org: **https://opengameart.org/content/isometric-hero-and-heroine**

2. Herramienta online para dibujar en cuadrícula. Disponible en: craftdesignonline.com: **https://craftdesignonline.com/pattern-grid/**
3. Herramienta online para crear diagramas UML. Disponible en go.gliffy.com: **https://go.gliffy.com/go/html5/launch**

Para la realización del videojuego

1. Dream Raid (Cinematic Action Soundtrack) por Matthew Pablo. Disponible en www.opengameart.com: **https://opengameart.org/content/dream-raid-cinematic-action-soundtrack**
2. Sword Slash Attack por qubodup. Disponible en www.freesound.org: **https://freesound.org/people/qubodup/sounds/184422/**
3. Melee Weapon Swings por Vull. Disponible en www.freesound.org: **https://freesound.org/people/Vull/sounds/215160/**
4. 50 RPG sound effects por Kenney. Disponible en www.opengameart.com: **https://opengameart.org/content/50-rpg-sound-effects**
5. 512 Sound Effects (8-Bit Style) por SubSpaceAudio. Disponible en www.opengameart.com: **https://opengameart.org/content/512-sound-effects-8-bit-style**
6. Fantasy Sound Effects Library por Little Robot Sound Factory. Disponible en www.opengameart.com: **https://opengameart.org/content/fantasy-sound-effects-library**
7. RPG Sound Pack por artisticdude. Disponible en www.opengameart.com: **https://opengameart.org/content/rpg-sound-pack**
8. 51 UI sound effects (buttons, switches and clicks) por Kenney. Disponible en www.opengameart.com: **https://opengameart.org/content/51-ui-sound-effects-buttons-switches-and-clicks**
9. Axe Impact 3 por LiamG_SFX. Disponible en www.freesound.org: **https://freesound.org/people/LiamG_SFX/sounds/322175/**
10. Axe Impact 2 – Pich Down por LiamG_SFX. Disponible en www.freesound.org: **https://freesound.org/people/LiamG_SFX/sounds/322172/**
11. Axe Impact – Gore por LiamG_SFX. Disponible en www.freesound.org: **https://freesound.org/people/LiamG_SFX/sounds/322171/**

12. Sword Slice for LiamG_SFX. Disponible en www.freesound.org: **https://freesound.org/people/LiamG_SFX/sounds/334238/**
13. Ladder por DasDeer. Disponible en www.freesound.org: **https://freesound.org/people/DasDeer/sounds/161809/**
14. Booms por studiomandragore. Disponible en www.freesound.org: **https://freesound.org/people/studiomandragore/sounds/401630/**
15. Sand Strom por gallifreanbuccaneer. Disponible en www.freesound.org: **https://freesound.org/people/gallifreyanbuccaneer/sounds/426560/**
16. Broom sweep por semccab. Disponible en www.freesound.org: **https://freesound.org/people/semccab/sounds/154376/**
17. Sci-fi_short_error por melissapons. Disponible en www.freesound.org: **https://freesound.org/people/melissapons/sounds/176238/**
18. Error_01 por distillerystudio. Disponible en www.freesound.org: **https://freesound.org/people/distillerystudio/sounds/327738/**
19. 004-driver_trouble.wav por wertstahl. Disponible en www.freesound.org: **https://freesound.org/people/wertstahl/sounds/409258/**
20. Dnb 110 BREAK3 3 por DOSCURREAL. Disponible en www.freesound.org: **https://freesound.org/people/DOCSURREAL/sounds/8325/**

Anexos

Glosario

- **RPG**: siglas de Role Playing Game. Juego de rol.
- **PixelArt**: estilo artístico en el que las obras se editan a nivel de píxel.
- **Smartphone**: teléfono inteligente.
- **Tablet**: computadora portátil de mayor tamaño a un smartphone.
- **Tabletop**: juego de mesa
- **ARPG**: siglas de Action Role Playing Game. RPG orientado a una experiencia más activa.
- **JRPG**: siglas de Japanese Role Playing Game. RPG de estilo japonés.
- **SRPG**: siglas de Strategy Role Playing Game. RPG centrado en decisiones estratégicas.

- **TRPG**: siglas de Tactical Role Playing Game. RPG centrado en el control de varias unidades y conocimiento del campo de batalla.
- **Arcade**: género o subgénero de juegos que plantean, en cada partida, un reto igual o similar hasta que el jugador sea capaz de superarlo.
- **Roguelike**: dungeon crawler que restringe el movimiento a una malla y con combate por turnos.
- **Roguelite**: dungeon crawler en el que el movimiento no está restringido a una malla y el combate es en tiempo real.
- **From scratch**: Hacer algo desde cero..
- **Input**: entrada al sistema a través de periféricos, por ejemplo, pulsar un botón del ratón.
- **Sprite**: imagen en 2D que define un personaje u objeto en un momento determinado.
- **Spritesheet**: agrupación de sprites. Normalmente por personajes u objetos.
- **Tile**: imagen en 2D que define una parte del terreno.
- **Tileset**: agrupación de tiles. Normalmente por biomas o zonas.
- **Nine-slice box**: técnica de creación de gráficos en los que una imagen cuadrada se divide en nueve secciones que permiten redimensionar la imagen sin perder nitidez.
- **Script**: fragmento de código.
- **Path**: camino que un objeto del juego recorrerá.
- **Timeline**: guion que define una secuencia de animaciones.
- **Shader**: programa que renderiza un objeto para mostrarlo por pantalla.
- **POO**: siglas de Programación Orientada a Objetos. Un modelo de programación en el que una clase, con sus variables y métodos define un objeto.
- **BOO**: implementación de la programación orientada a objetos en Unity.
- **ECS**: siglas de Entity Component System. Un modelo de programación en el que se separan variables de lógica.
- **Game design**: campo de estudio del diseño de videojuegos. Comprende tanto aspectos visuales y auditivos como mecánicas.
- **Gameplay**: jugabilidad de un juego. La experiencia de jugar.
- **Frame**: puede referirse a:
    - **Frames de una animación**: cada una de las subimágenes que componen la animación.

    - **Frames en una pantalla**: la imagen completa de la pantalla. Normalmente conlleva el cálculo de la lógica necesaria.
- **Framerate**: tasa de frames que una pantalla puede mostrar en un segundo. Depende de la potencia del procesador.
- **Bullet hell**: género o subgénero de algunos videojuegos en los que aparece una ingente cantidad de proyectiles enemigos por pantalla.
- **IDE**: siglas de Integrated Development Environment. Software que conforma un entorno y que permite gestionar y programar código.
- **Heap**: gran área común de memoria libre. El tamaño puede ser determinado en tiempo de ejecución y el tiempo de vida no depende ni del procedimiento actual ni del marco de la pila.
- **Stack**: región de la memoria en la que se almacena de manera local el estado de una función cuando se va a ejecutar. Sigue un método LIFO (*last-in first-out*)
- **GDD**: siglas de Game Design Document. Documento en el que se detallan todos los requisitos y como se va a desarrollar un videojuego.
- **Look and Feel**: conjunto de sensaciones que transmite un juego o animación a través del apartado gráfico.
- **Garbage Collector**: sistema implementado en algunos lenguajes de programación como C# o Java. Se encarga de gestionar la memoria a espaldas del desarrollador.
- **Creative Commons**: conjunto de instrumentos jurídicos que facilitan y regulan el uso y la compartición de cultura.
- **Level of Detail**: técnica de optimización que consiste en reducir el detalle de objetos lejanos. El detalle puede referirse a los polígonos de un modelo 3D o la tasa de ejecución de la lógica de un objeto.
- **Culling**: técnica de optimización que consiste en suprimir objetos.
- **Keyframe**: frame de una animación que define momentos críticos de una animación.
- **In-Between frame**: frame de una animación cuyo cometido es realizar una transición de un keyframe a otro.
- **Retweet:** redifusión de un Tweet escrito por otra persona desde nuestra cuenta.
- **Feed (de Twitter):** página principal de un perfil de Twitter, en el que se muestran todos los Tweets, *retweet* y *likes* de gente a la que un usuario sigue.